Escuela
Politécnica
Superior

App de asistencia en exteriores a personas con discapacidad visual bajo cualquier condición de luz

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Author:
Marcos Moreno Martínez
Mentor/s:
Miguel Ángel Cazorla Quevedo
Francisco Gómez Donoso

Julio 2019

Universitat d'Alacant
Universidad de Alicante

App de asistencia en exteriores a personas con discapacidad visual bajo cualquier condición de luz

Autor
Marcos Moreno Martínez

Tutor/es
Miguel Ángel Cazorla Quevedo
Ciencias de la Computación e Inteligencia Artificial (CCIA)
Francisco Gómez Donoso
Ciencias de la Computación e Inteligencia Artificial (CCIA)

Grado en Ingeniería Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2019

Agradecimientos

En primer lugar, agradecer a mis dos tutores la gran paciencia y tiempo invertido que han necesitado para guiarme con este proyecto, así como su predisposición y el equipo facilitado para poder llevar a cabo algunos de los puntos de este, sin dejar de lado los conocimientos que me han aportado durante las conversaciones que hemos tenido. También agradecer a los demás miembros de Rovit que me ayudaron con la adquisición de conocimientos acerca de herramientas que desconocía.

A mi familia, ya que son la razón de que haya podido cursar estos cuatro años de carrera. En especial a mis abuelos, pues son mi referencia desde pequeño, ya que han llevado una vida de trabajo duro y superación constante hasta el último momento, además de que me han inculcado una serie de valores por los que les estoy muy agradecido.

A mis amigos, tanto a los de toda la vida como a los que he tenido el privilegio de conocer a lo largo de la carrera. Agradecer en especial a aquellos que colaboraron enviándome algunas imágenes para la elaboración del *dataset* y que me aportaron algunos conocimientos acerca de Android, así como a aquellos que me acompañaron durante las caminatas buscando semáforos, que se que no lo pasaron muy bien.

Agradecer finalmente a todos los profesores del grado, pues sin haber aprendido de cada uno de ellos, este trabajo habría sido imposible. En especial, agradecer a Paco Pujol, el que acabaría siendo mi tutor académico y un gran apoyo durante los últimos tres años.

Índice general

Índice de figuras

Índice de cuadros

Glosario

CNN Convolutional Neural Network. 1, 11, 14, 16, 20, 26, 63

IDE Entorno de Desarrollo Integrado. 30

ILSVRC ImageNet Large Scale Visual Recognition Competition. 16, 17, 19

IOU Intersection over Union. 28, 29, 46–48, 50–52, 54

NaN Not-A-Number. 45–47

1 Introducción

Hoy en día, los *smartphones* son más que un instrumento útil de comunicación, pues se han vuelto una verdadera ventana entre la persona y el mundo entero, brindándole una nueva esfera para hacer negocios y relacionarse, ubicarse y obtener información sobre navegación con ayuda de GPS, e incluso de ocio bastante potente.

Existen múltiples aplicaciones para nuestros *smartphones* que utilizamos a diario de manera trivial, manejando sus menús sin dificultad, las cuales nos brindan múltiples facilidades de uso e incluso opciones de ayuda. Esto no se cumple para todo el mundo, pues por ejemplo, a las personas con discapacidad visuales les puede llegar a resultar imposible utilizar algunas de estas aplicaciones.

Es por esto que en los últimos años, algunas de estas aplicaciones han comenzado a integrar opciones para que a estas personas les sea más llevadero el utilizar sus *smartphones* para algo más que realizar llamadas, incluso se han desarrollado algunas aplicaciones enfocadas especialmente para ellos con fines de leerles textos manuscritos, darles información del entorno e incluso para manejarse por los menús del teléfono mediante comandos de voz.

El objetivo de este proyecto nace con vistas a seguir los pasos de estas aplicaciones, es decir, con el objetivo de desarrollar una aplicación capaz de brindar una cierta autonomía a los usuarios con discapacidad visual de *smartphones*, concretamente Android. Este proyecto se centra específicamente en la problemática que le supone a estas personas el cruzar la calle por un paso de peatones señalizado por semáforos sin sonido.

Para abordar este problema se emplearán métodos de aprendizaje profundo o deep learning. En concreto se hará uso de una CNN llamada YOLOv3, la cual será entrenada con un *dataset* propio que contempla situaciones lumínicas tanto por la noche como por el día, con el objetivo de que la aplicación sea operativa en cualquier momento. Este detector se introducirá en un servidor web que estará continuamente a la espera de recibir una imagen, la cual se enviará desde una aplicación Android con una interfaz sencilla, adaptada lo máximo posible para que personas con discapacidad visual sean capaces de utilizarla sin problema.

A lo largo de este documento se expondrán una serie de conceptos necesarios para

entender correctamente las aportaciones que supone este proyecto, las cuales se explicarán en detalle y se narrarán una serie de experimentaciones y procesos por los que hubo que pasar para llegar a conseguir finalmente una aplicación funcional, capaz de cumplir con el objetivo principal con el que se partió.

Mientras que en el capítulo 2 se pueden leer los objetivos principales de este proyecto, en el capítulo 3 se habla acerca del estado del arte de las aplicaciones que encontramos hoy día en la tienda Google Play enfocadas a personas con discapacidad visual. Pasando al capítulo 4, se puede leer acerca de algunos conceptos y tecnologías útiles para comprender algunas partes de este trabajo, es decir, el capítulo 4 contiene un pequeño marco teórico. A lo largo del capítulo 5 se tratan las tecnologías específicas que se aplicaron para poder desarrollar este trabajo. Durante el capítulo 6 se narran las aportaciones que suponen este proyecto en su conjunto, la fase de experimentación y resultados se contempla en el apartado 7, y finalmente en el capítulo 8 se exponen las conclusiones finales a las que se llegaron tras toda la fase de desarrollo del proyecto.

2 Objetivos

El objetivo principal en este proyecto es desarrollar una aplicación móvil capaz de dotar de cierta autonomía a una persona con algún tipo de discapcidad visual, permitiéndole cruzar la calle por un paso de peatones señalizado en cualquier momento del día y bajo cualquier condición de luminosidad. Este objetivo general se puede descomponer en los siguientes subjetivos:

- **Obtención de un *dataset* que contenga pasos de peatones y semáforos peatonales en sus dos estados posibles, tanto por el día como por la noche.**
- **Obtención de un modelo inteligente capaz de detectar y clasificar en una escena pasos de peatones y semáforos peatonales diferenciado sus dos posibles estados (verde y rojo).**
- **Programación de una aplicación Android que integre de alguna manera la capacidad de comunicarle al usuario si puede cruzar o no la calle mediante sonidos y vibraciones.** La interfaz de la aplicación será lo más sencilla y cómoda posible para el usuario, teniendo en cuenta sus limitaciones en todo momento. El usuario lo único que tendrá que hacer es capturar una imagen de su entorno y la aplicación hará el resto.
- **Desarrollo de un servidor web que atienda las peticiones de la aplicación móvil, procesando las imágenes que recibirá de esta y devolviendo una respuesta.**

3 Estado del arte

Cuando utilizamos nuestro *smartphone* Android, no somos conscientes en que lo que consideramos acciones triviales, como pulsar un menú o abrir una aplicación, para otros pueden suponer acciones con bastante dificultad. Es por esto que Google tiene una pequeña guía de accesibilidad en Android con diversos ajustes que se pueden activar en nuestros dispositivos si tenemos alguna discapacidad.

Además de esto, existen algunas aplicaciones que podemos descargar en el móvil que ayudan a personas esta discapacidad a utilizarlo de manera más sencilla y rápida, brindándoles una autonomía que quizás antes no tenían. Gracias a esto, las nuevas tecnologías móviles han pasado de ser vistas como barreras para la inclusión de personas con discapacidad, a mejorar su calidad de vida y participación en la sociedad.

Este tipo de aplicaciones enfocadas a solucionar dificultades y problemas típicos a los que se enfrentan las personas con discapacidad visual en su día a día suelen ser de acceso gratuito y hay múltiples enfoques. A lo largo de este capítulo se hablará de algunas de las aplicaciones más importantes que se pueden encontrar de manera gratuita en la tienda de Android Google Play [3].

Una de las aplicaciones que más ayudan es **Google Brailleback**, que se trata de una app de software libre desarrollada por Eyes-Free Project. Esta permite conectar al dispositivo móvil una pantalla braille a través de bluetooth y utilizarla junto al lector **Google Talkback** para ofrecer un servicio de voz y braille. Google Talkback es una audioguía del sistema para personas con problemas de visión, con comentarios hablados de cada menú y vibración para navegar por Android.

Voice Access es también otra de estas app, pero más que enfocada a personas con discapacidad visual, lo está para personas con parálisis o dificultad de movimiento. Permite acceder a cualquier elemento de la pantalla a partir de voz, y se puede lanzar con un "OK, Google".

También hay aplicaciones que son capaces de convertir el texto de artículos a voz, como **TTS** (Text to Speech) o **Linguoo**. Linguoo empezó siendo una app para leer páginas web a personas con discapacidad visual, pero ahora se una aplicación inclusiva que permite generar *playlists*, descargar todo tipo de artículos en voz y explorar noticias.

En este pequeño mundo de aplicaciones inclusivas, las personas que no vivimos con una discapacidad visual también podemos colaborar. Una de las aplicaciones mejor valoradas es **BeMyEyes**, pues permite a estas personas realizar videollamadas con voluntarios que les describen el medio en el que están o les ayudarán con problemas de guiado.

También hay aplicaciones enfocadas a mejorar la movilidad de estas personas por la ciudad, como *Map4all*, que pretende ser una especie de recopilador para la accesibilidad, recogiendo aquellos puntos con problemas de movilidad, auditivos, semáforos sin sonido o zonas sin acceso para sillas de ruedas. Esta no es solo una app enfocada a personas con discapacidad visual, si no a personas con múltiples minusvalías. **Lazarillo GPS** es también una aplicación muy completa que utiliza el GPS e informa con todo detalle de rutas, entornos, tiendas y servicios cercanos.

Una de las aplicaciones más completas actualmente en uso y actualización continua es **Eye-D**, una app capaz de informar al usuario de su ubicación, explorar y navegar por lugares de interés cercanos, evaluar el entorno con la cámara de su *smartphone* y leer texto impreso.

En este campo no solo ayudan todas estas aplicaciones especializadas, si no que otras como *App&Town* que de primeras no están dirigidas explícitamente a personas con discapacidad visual, incorporan guías enfocadas para ellas.

En la tienda de Android Google Play también se encuentra un juego llamado *A Blind Legend*, un regalo en el que no tenemos gráficos y todo se desarrolla a partir de experiencia auditiva.

Hasta ahora, como podemos ver, hay algunos proyectos relacionados con facilitar a personas con discapacidad visual a cruzar la calle, como los mencionados que utilizan GPS o **Button by Neatebox**, que busca que estas personas puedan pulsar los botones de cambio de estado de los semáforos para cruzar por los pasos de peatones desde sus *smartphones* o desde un *smartwatch*.

En cuanto al problema de cruzar la calle por un paso de peatones señalizado con semáforos, la única opción de hacerlo con seguridad total sería utilizando la app **BeMyEyes**, y teniendo que esperar a que en ese momento haya algún voluntario. Es por esta razón que en este proyecto se busca solucionar dicho problema, desarrollando una app que mediante el uso de la cámara del móvil y conexión a Internet el usuario sea capaz de cruzar la calle, sin depender de nadie más que de sí mismo.

4 Marco teórico

4.1. Deep Learning

Cuando se habla de *deep learning*, suele surgir la controversia de qué lo diferencia del *machine learning*, a pesar de que a día de hoy no parece existir ninguna definición clara que convenza a todos los investigadores que trabajan en este campo. Aunque el consenso mayoritario es que el *deep learning* no es más que una rama del *machine learning*, el cual consiste en hacer uso de herramientas más sofisticadas y complejas, que dotan a los sistemas de una capacidad de aprendizaje en la que el programador interviene de manera mínima.

La Figura 4.1 ofrece una buena representación gráfica de lo dicho anteriormente; tanto *machine learning* como *deep learning* surgen dentro del campo de la inteligencia artificial, pero a partir de cierto punto, dentro del *machine learning* se empiezan a emplear nuevas técnicas para automatizar el aprendizaje de manera más efectiva que los tradicionales métodos. Una de estas herramientas son las unidades lineales rectificadoras (ReLu), que son unas funciones de activación utilizadas en los rectificadores de las redes neuronales profundas, por poner un ejemplo.

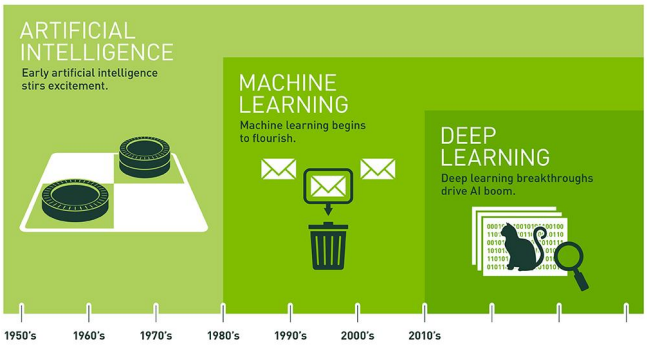


Figura 4.1: Relación temporal entre la Inteligencia Artificial, *Machine Learning* y *Deep Learning*. (Fuente: https://blogs.nvidia.com/wp-content/uploads/2016/07/Deep_Learning_Icons_R5_PNG.jpg.png)

Esta nueva rama se centra en simular el enfoque de aprendizaje que los seres humanos utilizamos para obtener ciertos tipos de conocimiento. Es por esto que el *deep learning* puede considerarse como una forma de automatizar el análisis predictivo [4].

Según Aaron Chavez, ex-Jefe Científico en AlchemyAPI, "la idea general de *deep learning* es usar redes neuronales para crear múltiples capas de abstracción para solventar un problema semántico complejo".

Es por esto que podríamos definir el *deep learning* como el área de la computación que mejora los procesos de, por ejemplo, la visión por computador y procesamiento de lenguaje, con el fin de resolver problemas en los que intervienen datos no estructurados, haciendo uso de redes neuronales.

4.2. Redes Neuronales

Las redes neuronales artificiales [5] son un modelo computacional inspirado en el comportamiento de su homólogo biológico. Consiste en un el conjunto de unidades computacionales llamadas neuronas artificiales, que están conectadas entre sí para transmitir una serie de señales que se someten a diversas operaciones, produciendo finalmente un valor de salida.

Cada una de estas neuronas está compuesta por una serie de señales de entrada ponderadas, una función de integración (consistente en una suma ponderada y una función de activación) y una señal de salida. Además de esto, las neuronas tienen un sesgo (*bias*) que puede considerarse como una entrada que siempre tiene el valor 1.0, pero que también debe ponderarse. Como aclaración, en caso de que una neurona tenga tres entradas, se requerirían cuatro pesos; tres para las entradas y uno para el sesgo.

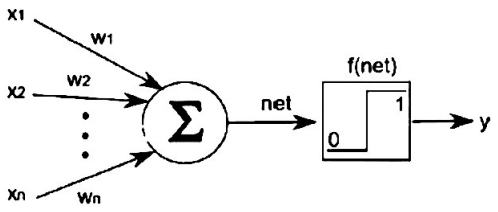


Figura 4.2: Esquema gráfico de una neurona artificial

En lo que concierte a los pesos, suelen ser inicializados de manera aleatoria con valores pequeños, en el rango de 0.3 a 0, aunque se pueden usar otros esquemas de inicialización más complejos. El porqué de dar valores pequeños a los pesos es debido

a que los pesos grandes indican una mayor complejidad y fragilidad del modelo, por lo que empezar con estos pesos muy grandes haría que costase mucho modificar y corregir el aprendizaje. Es por esto que es deseable mantener los pesos en la red con valores bajos y utilizar técnicas de regularización.

Estos pesos en los enlaces entre neuronas pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. Del mismo modo, a la salida de la neurona, existe una función de activación (limitadora o umbral), que modifica el valor resultado o impone un límite que se debe sobrepasar antes de propagarse la señal a otra neurona.

Las neuronas están organizadas en redes, donde cada una de las filas de neuronas que la componen se denominan capas (*layers*), pudiendo haber varias. La configuración más simple, es decir, una neurona con una serie de entradas y una salida, se denomina perceptrón simple.

En la Figura 4.3 podemos observar la arquitectura de una red de neuronas, donde cada una de las circunferencias se corresponde con una neurona. Como se puede observar, en este ejemplo hay cuatro capas de neuronas; la primera (la capa roja) se corresponde a la capa visible (o entradas), que son las que reciben los estímulos provenientes del exterior del sistema, es decir, se encargan de obtener y transmitir la información en bruto. Las neuronas negras se corresponden a los sesgos de cada neurona para cada una de las distintas capas. Las neuronas verdes son las que componen las dos capas ocultas de este caso particular, y aquí es donde se produce cualquier tipo de representación interna, sin tener contacto directo con la información de la entrada y la salida. Por último, las capas azules se corresponden con las neuronas de salida, que son las encargadas de proporcionar una respuesta al sistema.

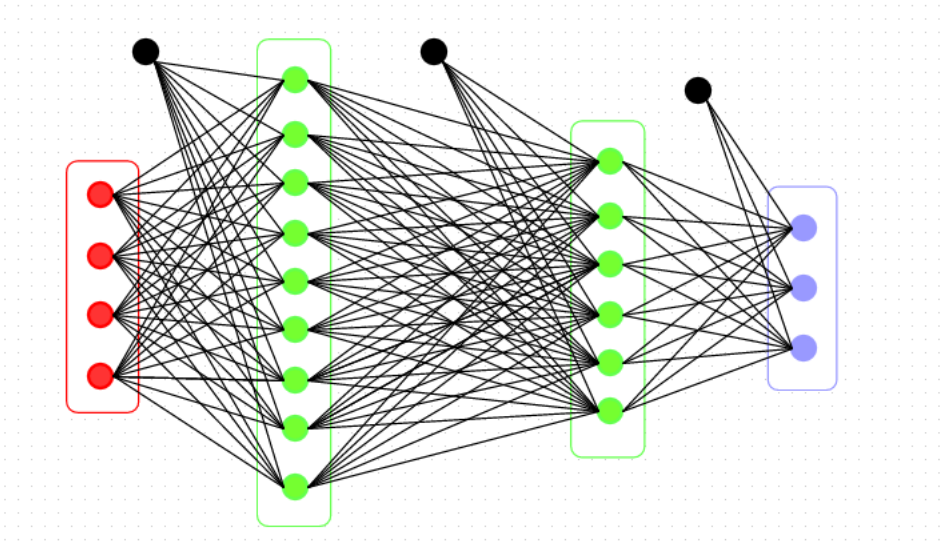


Figura 4.3: Esquema gráfico de red neuronal artificial. (Fuente: https://miro.medium.com/max/1400/1*C1WssIiKpRsfjKrle9IyMQ.png)

El número de neuronas de entrada de una red vienen determinadas por la forma de representación de la información del *dataset*, pero para que una red funcione también es preciso determinar el número de capas ocultas de la red, el número de neuronas en cada una de estas capas, y qué pesos se establecerán entre cada par de capas.

Para determinar el número de capas y de neuronas por capa, se suele hacer de manera aleatoria, por prueba y error normalmente, y teniendo en cuenta que a pesar de que con un número mayor de neuronas se pueden resolver problemas más complejos, también es verdad que se incrementa mucho el tiempo de entrenamiento.

En cuanto al proceso para determinar el valor de los pesos, se trata de un problema que se puede resolver de manera automática mediante un proceso de entrenamiento, que consiste en recopilar una serie de ejemplos de entradas e indicarle la salida que deseamos para cada una de estas muestras. En el caso particular de este proyecto, a la red lo que se le pasa es una serie de imágenes con semáforos peatonales y pasos de peatones, así como la información de donde se posicionan exactamente en la imagen. El algoritmo que se emplea para este proceso de reajuste de pesos se denomina *backpropagation* [6].

El método de*backpropagation* (o entrenamiento hacia atrás) es un método que se perfeccionó en la década de los 80. Este sistema de entrenamiento consiste en:

- Seleccionar unos pesos sinápticos, por lo general, de manera aleatoria.
- Introducir unos datos de entrada elegidos al azar de entre todos los datos de

entrada que se van a usar en el entrenamiento.

- Dejar que la red genere un vector de datos de salida (propagación hacia delante).
- Comparar la salida de la red con la deseada.
- La diferencia obtenida entre la salida y la deseada se denomina error, y se usa para ajustar los pesos de las neuronas de las capas de salida.
- El error se va propagando hacia atrás *(backpropagation)*, hacia la capa de neuronas anterior, y se usa para ajustar los pesos sinápticos en esta capa.
- Se continúa propagando el error hacia atrás y ajustando los pesos hasta que se alcance la capa de entradas.

Esta técnica requiere el uso de neuronas cuya función de activación sea continua, y por lo tanto, diferenciable. Es por esto que generalmente la función utilizada es de tipo sigmoidal como la que se muestra en la Figura 4.4.

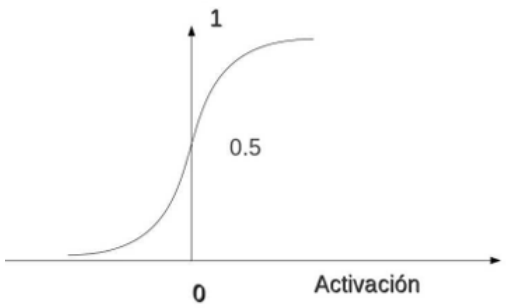


Figura 4.4: Gráfica de una función sigmoide.

La importancia del *backpropagation* recae en su capacidad de autoadaptar los pesos de las neuronas de las capas ocultas para aprender la relación que existe entre el conjunto de valores de entrada y sus salidas pertinentes. La red debe encontrar una representación interna que le permita generar las salidas deseadas cuando se le dan entradas de entrenamiento, y que pueda aplicar, además, a entradas no presentadas durante la etapa de aprendizaje.

En caso que la red no reconozca correctamente datos que no se encontraban en el conjunto de entrenamiento, estaremos hablando de un posible sobreentrenamiento de la red, es decir, la red se ha adaptado muy bien a los datos de entrenamiento, pero su configuración ya no sirve para reconocer y extraer información correcta de otros datos.

El problema con el sistema *backpropagation* es que es difícil encontrar los pesos para

la red de manera que la eficiencia sea máxima. Esto se debe a que la función que mide la eficiencia de las redes con capas ocultas, en la mayoría de casos, es una función compleja que representa muchos mínimos locales, y el mejor conjunto de pesos sinápticos es aquel que hace que la función alcance el mínimo global.

4.3. Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (CNN) [5] [7] son muy similares a las redes neuronales ordinarias, es decir, ambas se componen de neuronas con pesos y sesgos que deben aprender a ponderar. Estas neuronas reciben una serie de entradas, realizan un producto escalar y finalmente aplican una función de activación.

La principal diferencia de las CNN respecto a las redes neuronales comunes, es que las entradas que reciben son imágenes, lo que permite codificar ciertas propiedades de la arquitectura consiguiendo ganar en eficiencia, así como reducir parámetros y conexiones de la red.

Estas redes nacieron con la necesidad de poder procesar imágenes de manera efectiva y eficiente, resolviendo el problema de que las redes neuronales ordinarias no escalan bien para imágenes con una resolución razonable.

Supongamos un conjunto de datos de una imagen de 32×32 píxeles, la cual le pasamos a una red neuronal tradicional;esta requeriría 1024 pesos de entrada más el sesgo. Esto es manejable, aunque de manera muy justa, pero el aplanamiento de la matriz de la imagen de píxeles a un vector de valores de píxeles pierde toda la estructura espacial de la imagen. A no ser que todas las imágenes estén perfectamente redimensionadas, la red tendrá una gran dificultad a la hora de resolver el problema.

La ventaja con las CNNs es que preservan la relación espacial entre los píxeles mediante el aprendizaje de representaciones de características internas usando pequeñas matrices de datos de entrada. Las características se aprenden y se usan en toda la imagen, permitiendo que los objetos de las imágenes puedan estar en diferentes posiciones de la imagen y distintas escalas, y que la red los pueda detectar.Esto quiere decir que estas redes trabajan modelando de manera consecutiva pequeñas matrices de información, y luego combinan esta información en las capas más profundas de la red.

Una manera de entender esto es que las primeras capas irán detectando patrones o características simples, tal como bordes o sombras, mientras que las capas más posteriores tratarán de combinar todas estas formas simples en patrones más complejos, patrones de las diferentes posiciones de los objetos, iluminación, escalas, etc. Las capas finales harán coincidir una imagen de entrada con todos los patrones detectados

y proporcionarán una predicción final como una suma ponderada de todos los patrones detectados. En resumen, estos son algunos de los beneficios del uso de las redes neuronales:

- Requieren de un número menor de pesos para aprender respecto a las redes totalmente conectadas.
- Están diseñadas para ser invariantes a la posición del objeto y la distorsión de la escena.
- Aprenden y generalizan automáticamente las características del dominio de entrada.

En cuanto a la arquitectura de las CNNs, estas están compuestas por tres tipos distintos de capas:

- **Capas convolucionales**, que son las que aplican una gran cantidad de filtros convolucionales a las imágenes.
- **Capas de reducción** o ***pooling***, que se encargan de reducir la cantidad de parámetros al quedarse con las características más comunes.
- **Capa clasificadora totalmente conectada** o ***fully connected***, que proporciona el resultado final.

Estas redes neuronales están formadas por una secuencia alterna de capas convolucionales y capas de reducción, que irán sucesivamente reduciendo el tamaño de la información hasta ser manejable para ser trabajada por la última capa clasificadora totalmente conectada. En la Figura 4.5 se muestra un ejemplo de la arquitectura de una CNN.

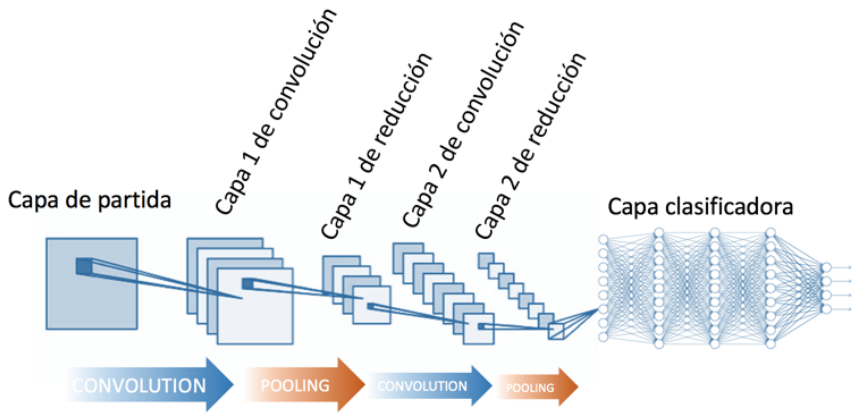


Figura 4.5: Ejemplo de esquema gráfico de una CNN. (Fuente: http://www.diegocalvo.es/wp-content/uploads/2017/07/red-neuronal-convolucional-arquitectura.png)

4.3.1. Capas convolucionales

Cada una de las capas convolucionales [8] está compuesta por una serie de *kernels* o filtros que nos devuelven un mapa de características de la imagen original, logrando reducir así el tamaño de los parámetros. En la Figura 4.6 se muestra un ejemplo de una operación de convolución en la que se observa cómo un parche de 3×3 de un filtro consigue reducir una imagen original de 7×7 en otra de 6×6.

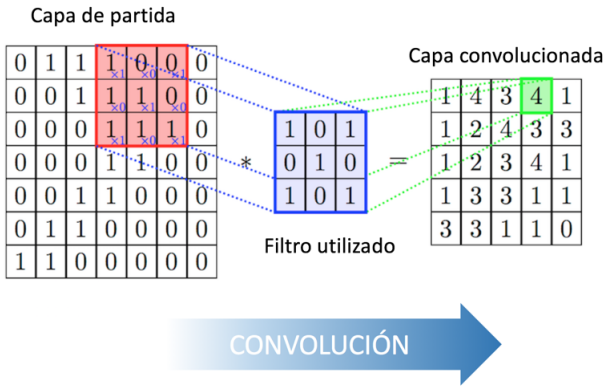


Figura 4.6: Ejemplo de aplicación de un filtro convolucional. (Fuente: http://www.diegocalvo.es/wp-content/uploads/2017/07/convoluci%C3%B3n.png)

La convolución aprovecha tres conceptos importantes que ayudan a mejorar cualquier

sistema de *machine learning*, que son:

- **Conectividad esparcida**: al aplicar un filtro de menor tamaño sobre la entrada original podemos reducir drásticamente la cantidad de parámetros. Se da la posibilidad de que las conexiones entre neuronas de estas redes no sean totales.
- **Parámetros compartidos**: al aplicar un filtro sobre toda la imagen, existen conexiones que se compartirán y se aplicarán sobre todos los valores de la imagen de entrada. Al compartir los pesos, se ayuda a mejorar la eficiencia del sistema.
- **Equivariante a la traslación**: este concepto significa que si se aplica alguna transformación de traslación a la imagen antes de ingresar a la CNN el resultado serán los mismos pero ordenados de otra manera. Este concepto se refuerza con la idea de *pooling*.

Algunos de los parámetros más importantes a tener en cuenta en las capas de convolución son:

- El **tamaño del filtro**, que es el tamaño del parche que va a ir recorriendo nuestra imagen, procesándola poco a poco.
- La **profundidad de la capa convolucional**, que es el número de filtros que se le va a aplicar a la imagen de entrada. La profundidad de una imagen equivale al número de filtros que le vamos a pasar en cada una de las convoluciones. En la Figura 4.6 se observa la aplicación de un filtro convolucional con un *kernel* de tamaño $5 \times 5 \times 3$ a una imagen con una profundidad de tres capas, dando como resultado final una imagen reducida tanto en ancho y alto como en profundidad.
- El ***stride* o paso** define cada cuantos píxeles se procesará un parche de la imagen, es decir, si se quiere procesar cada dos píxeles, avanzando de uno en uno... Esto afecta a la convolución, pues a mayor paso, más se reducirá la imagen.

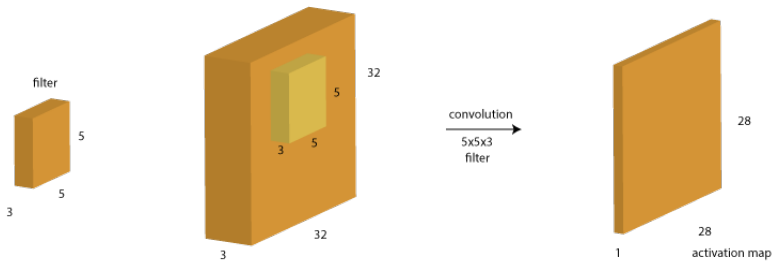


Figura 4.7: Aplicación de filtro convolucional de profundidad 3. (Fuente: https://relopezbriega.github.io/images/conv_layer.png)

4.3.2. Capas de *pooling* o de reducción

La capa de reducción o *pooling* se coloca de manera posterior a una capa convolucional. El concepto del *pooling* es muy sencillo a la par que importante; consiste en reducir las dimensiones del ancho y alto de la imagen sin afectar a la dimensión de profundidad. Esta operación también se conoce bajo el nombre de reducción de muestreo, ya que la reducción de la imagen lleva a una pérdida de información. Sin embargo, una pérdida de este tipo es beneficiosa para la red por dos razones; al reducir el tamaño se conduce a una menor sobrecarga de cálculo para las próximas capas de la red, y ayuda a reducir el sobreajuste durante el proceso de entrenamiento.

En *deep learning* el sobreajuste es el efecto de sobreentrenar un algoritmo de aprendizaje solo y exclusivamente con datos para los que conoce el resultado. El algoritmo de aprendizaje debe alcanzar el estado en el que sea capaz de predecir el resultado en otros casos a partir de lo aprendido durante el entrenamiento, pero cuando un sistema sobreentrena, el algoritmo de aprendizaje puede quedar muy ajustado a unas características específicas de los datos del conjunto de entrenamiento. Durante el proceso del sobreajuste lo que ocurre es que el algoritmo cada vez irá respondiendo mejor a las muestras del entrenamiento, mientras que se irá alejando de mantener una buena predicción con nuevas muestras externas al conjunto de entrenamiento.

Existen tres métodos de *pooling*:

- ***Max-pooling***: se escoge el mayor valor de entre todos los píxeles que caen dentro de un kernel determinado.
- ***Min-pooling***: se escoge el menor valor de entre todos los píxeles que caen dentro de un kernel determinado.
- ***Average pooling***: se calcula el valor medio de entre todos los píxeles que caen dentro de un kernel determinado.

En la Figura 4.8 se puede observar un caso gráfico de aplicación de *max-pooling* con un *kernel* de 2×2 reduciendo el ancho y alto de la imagen a la mitad.

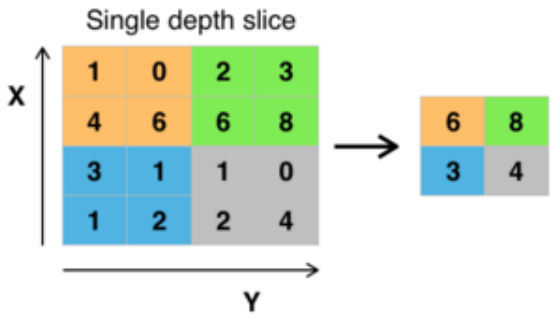


Figura 4.8: Ejemplo de aplicación de max-pooling. (Fuente: https://relopezbriega.github.io/images/Max_pooling.png)

4.3.3. Capa clasificadora totalmente conectada

Tras todas las capas convolucionales y de *pooling*, las CNNs utilizan generalmente capas completamente conectadas en la que cada píxel se considera como una neurona separada al igual que en una red neuronal regular. Esta última capa tendrá tantas neuronas como el número de clases que se debe predecir.

4.3.4. Arquitecturas CNN más relevantes

En esta sección se hará mención a algunas de las CNN más relevantes [9] y se expondrá por qué lo son, exponiendo su arquitectura y las aportaciones que implicaron en este campo. Las redes de las que se hablará son: LeNet por ser la primera CNN, VGGNet por ganar en 2014 a GoogleNet en la categoría de localización en la ImageNet Large Scale Visual Recognition Competition (ILSVRC), y ResNet por introducir el concepto de residuo.

4.3.4.1. LeNet

LeNet [10] [11] es la primera CNN propuesta en la historia que nace con el objetivo de identificar texto manuscrito. La arquitectura de esta red está compuesta por una capa de entrada que recibe una imagen de 28×28. Esta imagen es procesada por una primera capa convolucional que aplica un total de cuatro filtros de tamaño 5×5, generando cuatro mapas de características de tamaño 24×24. Estos mapas de características después pasan por una capa de *average pooling* de tamaño 2×2 que reduce los mapas a tamaño 12×12. Estos cuatro mapas pasar por otra capa de convolución, donde

a cada uno de ellos se le aplican cuatro convoluciones de tamaño 5×5, generando doce nuevos mapas de tamaño 8×8. Estos doce mapas pasan ahora por otra capa de *average pooling* de tamaño 2×2, reduciendo el tamaño de estos a 4×4. Finalmente, estos mapas alimentan a una red *fully connected*, cuya salida final es un grupo de nodos que clasifican el resultado. Esta arquitectura se puede observar en la Figura 4.9.

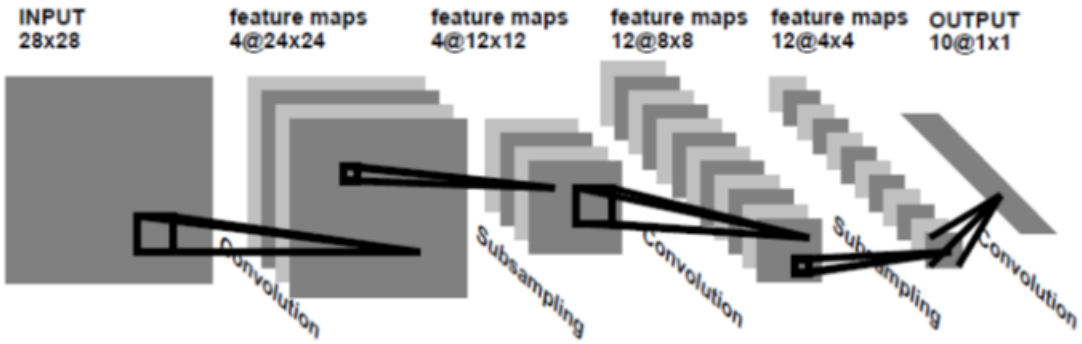


Figura 4.9: Arquitectura de LeNet-1. (Fuente: https://cdn-images-1.medium.com/max/1000/1*ge5OLutAT9_3fxt_sKTBGA.png)

Esta red fue propuesta en 1989, aunque en los años siguientes iría mejorándose. Las implementaciones más actuales de esta arquitectura modifican, por ejemplo, el uso de *average pooling* por el de *max-pooling*; también añaden más convoluciones dentro de las capas convolucionales y más capas neuronas en la *fully connected*, lo que mejora los valores de acierto durante la fase de test.

4.3.4.2. VGGNet

Como se comenta al principio de este subapartado, este modelo superó a GoogleNet en las pruebas de localización durante la ILSVRC en 2014, aunque la ganadora de la competición fue GoogleNet.

La arquitectura de VGGNet [12] [13] se caracteriza por su simplicidad; solo usa capas convoluciones de 3×3 unas encima de otras en profundidad creciente, para ir reduciendo los mapas de características hace uso de capas *max-pooling* y al final concluye en dos capas *fully connected*, cada una con 4096 nodos a los que les sigue un clasificador *softmax*. En la Figura 4.10 se observa una representación gráfica de esta arquitectura.

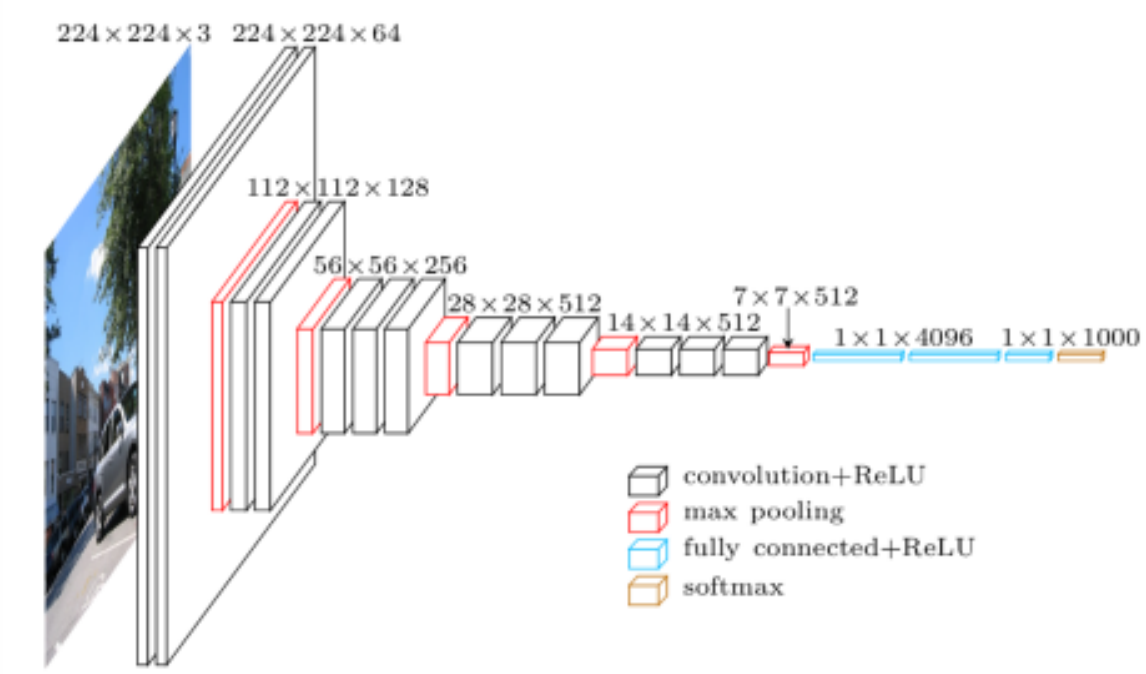


Figura 4.10: Arquitectura de VGGNet. (Fuente: http://jesusutrera.com/articles/img/vgg_model.png)

Los desarrolladores de esta red se encontraron con que era muy costoso entrenar este modelo (costaba mucho que convergiese cuanto más profunda era la red), por lo que desarrollaron pequeñas versiones de VGGNet con menos pesos con los que operar. Los pesos que obtuvieron con los entrenamientos de estas pequeñas redes fueron los que usaron para inicializar las versiones de VGGNet más profundas; este proceso se denomina pre-entrenamiento (*pre-training*).

Además de ser una red muy costosa de entrenar, los pesos son bastante grandes en términos de disco y ancho de banda, llegándose a superar los 574 MB en la versión más profunda. La combinación de estos dos problemas hace que la implementación de VGGNet sea muy pesada.

A pesar de estos inconvenientes, sigue utilizándose en muchos problemas de *deep learning* con clasificación de imagen, aunque son más deseables otras arquitecturas más pequeñas como GoogleNet o SqueezeNet.

4.3.4.3. ResNet

A diferencia de las arquitecturas de las redes anteriores, esta tiene una arquitectura compuesta por múltiples micro-arquitecturas llamadas "*building blocks*" [1] [14] (junto con sus capas de convolución, *pooling*, etc) como se puede observar en la Figura 4.11. Esta arquitectura se ha convertido en un trabajo seminal, demostrando que las redes extremadamente profundas pueden ser entrenadas utilizando SGD (Stochastic Gradient Descent) mediante el uso de módulos residuales.

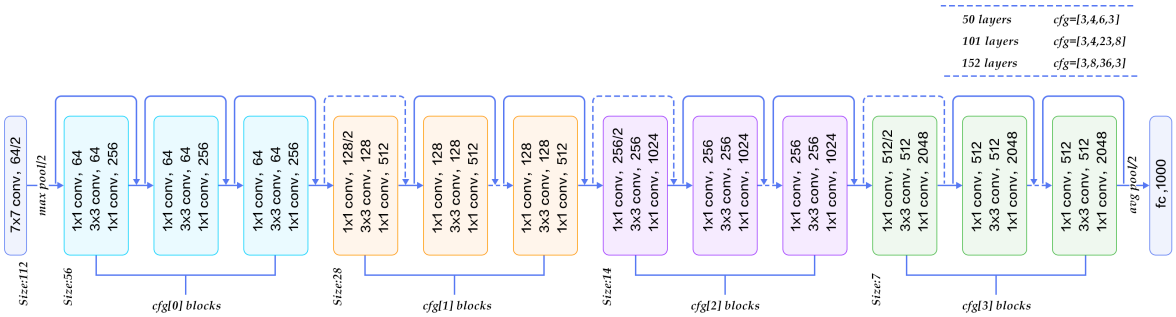


Figura 4.11: Arquitectura de ResNet. (Fuente: http://www.jesusutrera.com/articles/img/resnet.png)

Para resolver el problema de la desaparición de gradientes se agrega un salto para agregar la entrada x a la salida después de algunas capas de pesos, como se muestra en la Figura 4.12. Por lo tanto, la salida es $H(x) = F(x) + x$, y las caspas de peso son en realidad para aprender un tipo de mapeo residual $H(x) = F(x) - x$. En caso de que se produzca la pérdida de un gradiente, siempre se tiene la identidad x para transferir a las capas anteriores. Gracias a esta aportación del uso de residuos fue la ganadora de la ILSVRC en 2015.

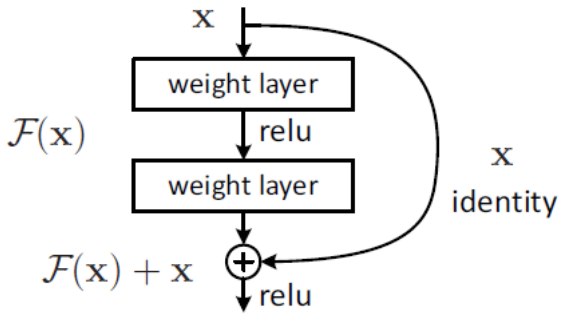


Figura 4.12: Ejemplo de *building block* de ResNet. (Fuente: [1])

5 Metodología

En este apartado se definirán las tecnologías específicas utilizadas a lo largo del proyecto, entrando en detalles de funcionamiento y conceptos, instalaciones, errores y soluciones. Se abordará el framework de desarrollo Darknet, la arquitectura de la CNN YOLOv3 considerada el estado del arte en el ámbito de la detección de objetos en tiempo real, servicios web y algunos conceptos importantes al programar en Android.

5.1. Darknet

[15] Darknet es un framework de desarrollo de redes neuronales de código abierto escrito en C y CUDA. Es rápido, fácil de instalar, y soporta el computo en CPU y GPU. Este es el framework utilizado para crear la arquitectura de la red neuronal YOLOv3 empleada en este proyecto.

Para instalar este framework en el equipo se siguieron los pasos descritos en la página oficial de Darknet [1]. El primer paso fue clonar el repositorio original en el equipo y hacer *make* para compilar los archivos que componen el framework. Para ello se siguen los siguientes pasos en una terminal:

```
git clone https://github.com/pjreddie/darknet.git
cd darknet
make
```

En este repositorio no solo se encuentra Darknet, si no que también contiene las distintas versiones de arquitecturas de YOLO. Una vez terminada la compilación de los archivos, para comprobar que se ha completado correctamente se puede probar a ejecutar el comando "./darknet", esperando el siguiente resultado en la salida:

```
usage: ./darknet <function>
```

[1] https://pjreddie.com/darknet/install/

5.1.1. Compilar Darknet con CUDA

Darknet funciona bien en la CPU, pero es mucho más rápido en la GPU. Para poder compilar con GPU es necesario que el equipo tenga una Nvidia GPU y tener instalado CUDA. En el caso particular de este proyecto se utilizaron dos GPUs distintas; una Nvidia Quadro P6000 y una Nvidia GeForce GTX 960M.

La GPU Nvidia Quadro P6000 fue facilitada por el equipo de investigación Rovit de la Universidad de Alicante. Esta GPU fue utilizada para entrenar distintos modelos de clasificadores, pues tiene una memoria de 24 GB y los tiempos de entrenamiento se reducen mucho respecto al uso de la GeForce GTX 960M, que es la que incorpora mi equipo personal, y tan solo tiene una memoria de GPU de 4 GB. Esta segunda, fue utilizada para las distintas pruebas y testeos del modelo. En el equipo de la Quadro P6000 se instaló la versión de CUDA 8.0 y en el de la GeForce GTX se usó CUDA 9.2.

Para evitar errores a la hora de usar CUDA, es aconsejable revisar el archivo .bashrc y comprobar si tenemos correctamente inicializados los *export* de CUDA. En el caso del equipo de Rovit se añadieron las siguiente líneas al final del .bashrc:

```
# CUDA Toolkit
export CUDA_HOME=/usr/local/cuda-8.0
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64:$LD_LIBRARY_PATH
export PATH=${CUDA_HOME}/bin:${PATH}
```

Una vez instalada CUDA, hay que modificar la primera linea del fichero Makefile ubicado en el directorio base. En esta primera línea se indica si se quiere hacer uso o no de la GPU. Si queremos hacer uso de ella, debemos indicar lo siguiente:

```
GPU=1
```

Para habilitar el uso de CUDA, es necesario volver a compilar con *make* el proyecto. Hay que tener en cuenta que por defecto la GPU en la que correrán los modelos será en la primera ubicada en el equipo, que corresponde a la posición '0'. En el caso del equipo de la Nvidia Quadro P6000, este incorporaba otra GPU. Para forzar el uso de la Quadro, una solución fue acceder al .bashrc y hacer que CUDA solo visualizase la Quadro ubicada en la posición '1' con la siguiente linea de código:

```
export CUDA_VISIBLE_DEVICES=1
```

5.2. YOLOv3

Las arquitecturas de las CNNs en sus orígenes con LeNet solo servían para hacer clasificación, pero estudios posteriores han ido demostrando que se puede aplicar este método para obtener otros resultados como regresión, segmentación a nivel de píxel o para obtener la localización de los objetos dentro de la imagen. Para este proyecto era de interés una arquitectura capaz de clasificar y localizar los objetos en la imagen, y YOLO cumple estos dos requisitos, pues devuelve un resultado que incluye el porcentaje de acierto, una etiqueta con el nombre del objeto y las dimensiones de un *bounding box* que encierra a los objetos detectados.

YOLO (*You Only Look Once*) [16] [17] [2] es un innovador enfoque para la detección de objetos aplicando *deep learning* y CNNs, distinguiéndose de sus competidores porque solo requiere de "ver" la imagen una sola vez (tal como su nombre indica), permitiéndole ser más rápido que otros detectores sacrificando un poco de exactitud.

Para llevar a cabo la detección, lo primero que hace es dividir la imagen en una cuadrícula de tamaño $S \times S$. Para cada una de las celdas predice 5 posibles *bounding boxes* y calcula la probabilidad de que cada una de ellas contenga un objeto, es decir, se calculan $S \times S \times 5$ *bounding boxes*, aunque la mayoría de ellas tendrán un nivel de probabilidad muy bajo. Después de obtener estas predicciones se procede a eliminar las cajas que estén por debajo de un límite aplicando un *non-max suppression*. Este proceso se puede observar en la Figura 5.1.

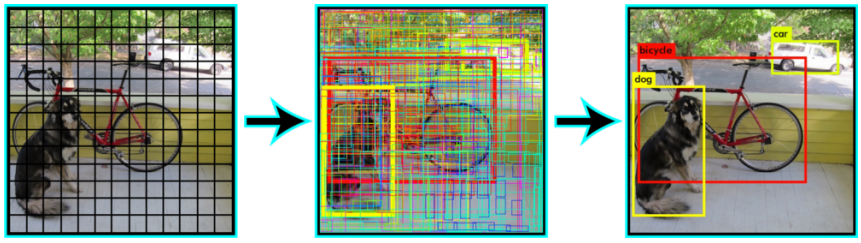


Figura 5.1: Predicción de *bounding boxes* y clases para cada celda de una imagen. (Fuente: https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246)

De manera conceptual, YOLO funciona así, pero a continuación se desglosará por bloques algunas cuestiones más específicas tales como la predicción de las bounding box, la predicción de clases, la predicción a distintas escalas y la arquitectura de la CNN.

5.2.1. Predicción de los bounding box

El sistema predice 5 *bounding boxes priors* (anclas) para cada una de las celdas del mapa de características de salida. La red predice cuatro coordenadas para cada uno de estos *priors*: t_x, t_y, t_w y t_h. Si la celda está desplazada desde la esquina superior izquierda una distancia c_x y c_y, y el bounding box prior tiene un ancho y alto p_w y p_h, entonces la predicción se corresponde con:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^(t_w) \\
b_h &= p_h e^(t_h)
\end{aligned}
$$

En la Figura 5.2 se puede apreciar de manera gráfica a que corresponde cada una de estas ecuaciones y las formas para las *bounding boxes* prior que se utilizan son las que muestra la Figura 5.3.

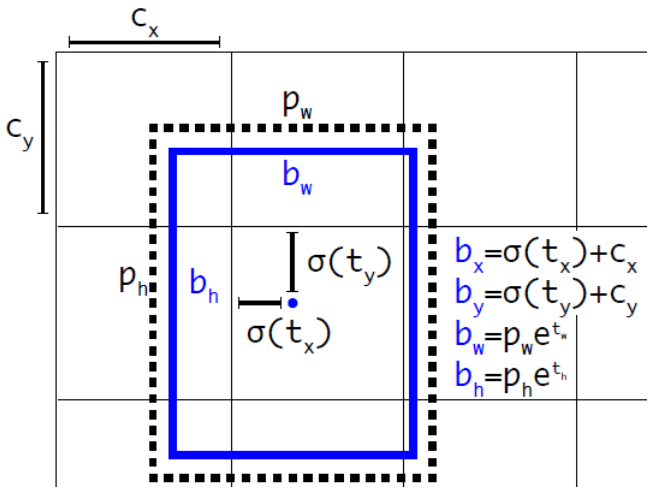


Figura 5.2: *Bounding boxes* con las dimensiones de prior (punteado) y predicción de ubicación (en azul). El algoritmo predice el alto y ancho del prior como *offsets* de los centroides de los *clusters*. La predicción del centro de coordenadas de las cajas se hace en relación con la localización de la aplicación del filtro usando una función de sigmoide. (Fuente: [2])

.

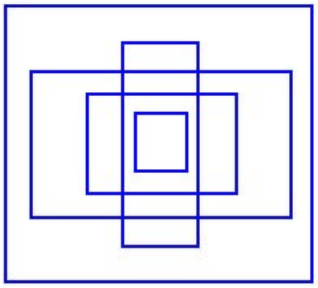


Figura 5.3: Formas de los cinco prior por defecto. (Fuente: https://zhanghanduo.github.io/post/yolo2/)

Como limitamos la predicción de la ubicación, la parametrización es más fácil de aprender, lo que hace a la red ser más estable. El uso de *clusters* de dimensión junto con la predicción directa de la ubicación central de la *bounding box*, mejora la red respecto a la versión anterior.

Durante el entrenamiento se usa la suma de la pérdida del error al cuadrado *(squared error loss)*. Si el *ground truth*, entendida como la "etiqueta" (etiqueta = clase del objeto + bounding box que la contiene) que debe reconocer y devolver el modelo (respuesta esperada) para alguna predicción de coordenadas es $\hat{t}_*$, nuestro gradiente es el valor del *ground truth* menos la predicción: $\hat{t}_* - t_*$. Este valor del *ground truth* es fácil de obtener para cada bounding box invirtiendo las ecuaciones anteriores.

YOLOv3 predice un *objectness score*, que sirve para diferenciar entre el fondo y los objetos, proporcionando a estos últimos un valor elevado utilizando una operación de regresión lógica. En caso de que el prior se solape con el *ground truth* más que cualquier otro prior, su *objectness* será de valor 1. En caso de que el *bounding box prior* no sea el que se solape mejor con el *ground truth* pero su valor de solape sea superior a uno fijado para un *threshold* (que por defecto es 0.5), se ignorará la predicción, lo que no implica ningún coste. YOLOv3 solo asigna un *prior* a cada uno de los *objetos ground truth*, y si a un *prior* no se le puede asignar un objeto *ground truth*, no incurrirá en ninguna pérdida de coordenadas o predicciones de clase, solo lo hará en el *objectness*.

5.2.2. Predicción de las clases

Cada uno de los priors predice las clases que es posible que contenga en su interior haciendo uso de un clasificador con múltiples etiquetas. Esta versión de YOLO no utiliza una función *softmax* ya que se dieron cuenta de que no es necesaria para un buen

rendimiento, y el lugar de ello, se hace uso de clasificadores logísticos independientes. Durante el entrenamiento se utiliza pérdida binaria de entropía cruzada *(binary cross-entropy loss)* para la predicción de las clases.

Este enfoque ayuda cuando tratamos con *datasets* complejos, en los que hay etiquetas superpuestas, como por ejemplo "persona" y "niño". Usando *softmax* se impone la suposición de que cada caja contiene una única clase, lo que sería falso en este caso. Al evitar usar la función *softmax* se reduce la complejidad del cálculo y se obtiene un mejor modelo multi-etiqueta.

5.2.3. Predicciones a través de escalas

YOLOv3 predice cajas a tres escalas diferentes. El sistema extrae características a estas escalas utilizando un concepto similar al de las redes piramidales de características (FPN). En esta versión se han añadido, respecto al extractor de características original, múltiples capas nuevas de convolución.

En cada una de estas tres escalas se llevan a cabo tres predicciones compuestas por una bounding box, un valor de objetividad *(objectness)* y X predicciones de clase (en el caso del entrenamiento original, X tiene un valor de 80 ya que fue entrenado con un *dataset* llamado COCO que contenía 80 clases diferentes), por lo que el tensor es $N \times N \times [3 \times (4 + 1 + 80)]$ para los cuatro *offsets* del *bounding box* que se predice por cada caja de anclaje, una predicción de objectness y las 80 predicciones de clase.

Estas tres predicciones las realiza en los siguientes puntos de la red:

1. La primera de estas predicciones la hace con el mapa de características resultante en la última de las capas convolucionales.

2. Para hacer la segunda predicción salta dos capas atrás y realiza una operación de *upsampling* x2. También se toma un mapa de características anterior al final y se fusiona con las características resultantes en la operación de *upsampling* usando concatenación. Este método permite obtener más información semántica significativa de las características a las que se les aplicó el *upsampling* e información más precisa del mapa de características anterior.

   Finalmente se aplican unas pocas capas convolucionales para procesar bien este nuevo mapa de características combinado y se predice un tensor, pero ahora con un tamaño duplicado.

3. Se repite el proceso anterior retrocediendo dos capas más. Con esto se consigue que la última predicción se beneficie de todos los cálculos previos, así como de

las características de grano fino desde el principio de la red.

Para determinar los *bounding box prior*, se hace uso de 9 *clusters* y 3 escalas arbitrarias. Estos 9 *priors* se agrupan en tres diferentes grupos conforme a sus escalas. Cada grupo se asigna a un mapa de características específico en la detección de objetos. Los 9 *clusters* usados en el *dataset* COCO son: (10x13), (16x30), (33x23), (30x61), (62x45), (59x119), (116x90), (156x198), (373x326).

5.2.4. Arquitectura de la red

La arquitectura de esta red neuronal está compuesta por sucesivas capas convolucionales de 3x3 y 1x1 que suman en total 53 capas. Esta CNN se llama *Darknet-53*, y es la expuesta en la Figura 5.4.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figura 5.4: Arquitectura de la CNN *Darknet-53*. (Fuente: [2])

5.2.5. Primeros pasos con YOLOv3

Para obtener las distintas arquitecturas y versiones de YOLO se debe descargar el repositorio original de Darknet mencionado anteriormente [2]. Una vez se cuenta con

[2]https://github.com/pjreddie/darknet

Darknet en el equipo, se pueden hacer algunas ejecuciones de prueba para comenzar a familiarizarse con el funcionamiento de la red.

En el sitio web oficial de YOLO [18] es posible descargar unos pesos ya preentrenados para YOLOv3, y para ello basta con ejecutar el siguiente comando en una terminal:

```
wget https://pjreddie.com/media/files/yolov3.weights
```

Partiendo de que la configuración de YOLOv3 se encuentra en el subdirectorio de Darknet "/cfg" y de que ya se tendrían unos pesos para el modelo, podemos ejecutar el detector con el siguiente comando:

```
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

Lo que indicamos en el anterior comando es que queremos hacer uso del detector, utilizando la configuración de YOLOv3, con los pesos recientemente descargados, y queremos aplicarlo sobre una imagen llamada *doj.jpg*. Lo que se observa tras la ejecución en terminal es lo que se muestra a continuación, mientras que la imagen resultante es la mostrada en la Figura 5.5, donde se observa que el algoritmo dibuja unos *bounding box* encima de todos los objetos detectados.

```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   608 x 608 x   3   ->   608 x 608 x  32  0.639 BFLOPs
    1 conv     64  3 x 3 / 2   608 x 608 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    ......
  105 conv    255  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 255  0.754 BFLOPs
  106 yolo
Loading weights from yolov3.weights...Done!
data/dog.jpg: Predicted in 1.709188 seconds.
dog: 100%
truck: 92%
bicycle: 99%
```

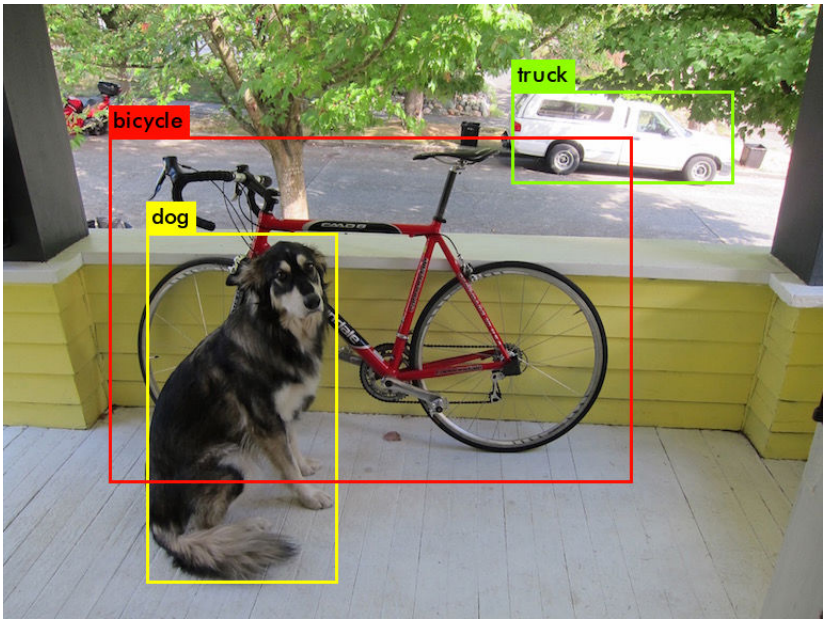


Figura 5.5: Ejemplo de resultado de una aplicación de YOLOv3 sobre una imagen.

Al usar el comando *detector*, lo que estamos haciendo realmente es utilizar una abreviatura del comando *detector test*. También se pueden usar estos comandos para detecciones en múltiples imágenes, es decir, es posible cargar una vez el modelo e ir pasándole imágenes múltiples veces. Para ello se emplea el mismo comando pero sin definir una imagen al final de este.

```
./darknet detect cfg/yolov3.cfg yolov3.weights
```

No solo existe el comando de *test*, si no que también está la posibilidad de utilizar *train*, *valid* y *recall*.

El comando *train* se utiliza para entrenar o reentrenar un modelo, pues es posible cortar un entrenamiento y continuarlo a partir de los pesos que se deseen posteriormente. Cuando se entrena un nuevo modelo, es aconsejable comenzar con los pesos previamente calculados para el modelo original de YOLOv3.

En caso de que ejecutemos el comando *valid*, lo que obtendremos será una serie de documentos *.txt* con el nombre de cada una de las clases que contiene el *dataset* del modelo, en cuyas líneas aparecerá el nombre de cada una de las imágenes que contienen un objeto de dicha clase y la posición en la que se encuentra dicho objeto en la imagen.

Por último, cuando se utiliza el comando *recall* obtenemos en terminal una serie de líneas, en las que se muestra un valor de *recall* e IOU medio. El valor de *recall* se

entiende como el porcentaje de objetos que se han detectado correctamente, es decir:

$$Recall = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ negativos}$$

El IOU es una métrica que se utiliza para determinar cómo de acertado es un modelo a la hora de detectar los objetos. El valor de IOU se calcula como la relación entre el área que se produce de la intersección de la *bounding box* de la etiqueta del objeto y la que predice el modelo, entre el área total que conforman estas. En la Figura 5.6 se puede apreciar de manera gráfica esta relación.

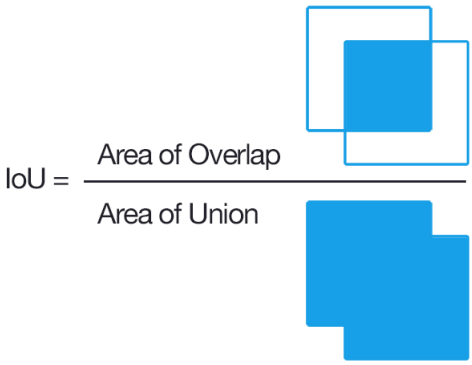


Figura 5.6: Representación gráfica de la obtención de IOU. (Fuente: https://timebutt.github.io/static/understanding-yolov2-training-output/)

5.3. Servidor web

Un servidor web [19] es un programa informático que procesa una aplicación del lado del servidor, realizando conexiones bidireccionales o unidireccionales y síncronas o asíncronas con el cliente y generando o cediendo una respuesta en cualquier lenguaje. El código recibido por el cliente es renderizado por un navegador web, y para la transmisión de datos suele utilizarse algún protocolo, que en el caso de este proyecto ha sido el protocolo HTTP.

Para este proyecto se hace necesario el uso de servidores web debido a que se pretende enviar una fotografía capturada en exteriores a un equipo con una GPU, pues un smartphone no es capaz de procesar una imagen usando un modelo de YOLO. El servidor lo utilizaremos para que el cliente pueda enviarle una imagen y este le responda al cliente en función de si considera que se puede o no cruzar la calle.

5.4. Android

Android [20] es un sistema operativo para dispositivos de pantalla táctil, el cual nos permite la distribución, instalación y ejecución de software dentro de él, con el nombre de 'aplicaciones móviles'. Para programar estas aplicaciones móviles se hace uso del Entorno de Desarrollo Integrado (IDE) oficial de Google llamado Android Studio. Con este IDE podemos programar en lenguaje Java, C++ y Kotlin, pero en este proyecto se hace uso del lenguaje Java.

Cuando programamos en Android, el primer concepto que debe quedar claro es qué es un *activity*, pues es lo más básico y más utilizado al desarrollas aplicaciones Android. Se puede decir que cada pantalla es una *activity*, es decir, si una aplicación tiene cinco pantallas, esta tendrá cinco *activities*.

Estas *activities* están conformadas por dos partes; la parte lógica y la parte gráfica. La parte lógica consiste en un archivo *.java* que es la clase que se crea para poder manipular, interactuar y colocar el código de esa *activity*. La parte gráfica es un XLM, que tiene todos los elementos vemos en una pantalla, declarados con etiquetas parecidas a las de HTML.

6 Aportaciones

En este capítulo se hablará de las tres fases principales en las que se puede dividir este proyecto. En primer lugar se explicará cómo se elaboró el *dataset* y se aportará información relevante a su uso. A continuación, se expondrá cómo se reajustó el modelo original de YOLOv3 para que se adaptase a nuestro problema, indicando todas las modificaciones, creaciones de nuevos archivos, reconfiguraciones, etc. Finalmente se explicará cómo de desarrolló la aplicación en Android para darle una utilidad al clasificador, así como de la elaboración de un servidor que atendiese las peticiones de la aplicación y le enviase una serie de respuestas.

6.1. Elaboración de *dataset* propio

Previamente a la elaboración del *dataset* propio, se hizo una investigación durante días sobre varios *datasets* ya elaborados que se pudieran adaptar al problema que tratábamos de resolver. Tras la revisión de varios *dataset*, ninguno de ellos cumplía lo que se buscaba, pues ninguno de ellos contenía pasos de peatones ni semáforos peatonales suficientes, además de que las pocas imágenes que podían servir no eran suficientes.

Tras finalizar este proceso de búsqueda, se concluyó que la mejor opción era comenzar con la elaboración de un *dataset* propio que no fuera muy amplio (alrededor de mil fotografías). Las fotografías de este *dataset* debía contener tantas fotos de noche como de día para que el algoritmo pudiese aprender a distinguir tanto pasos de peatones como semáforos peatonales (tanto en verde como en rojo) en cualquier momento del día y bajo cualquier condición de luz.

La mayoría de imágenes que contiene este *dataset* son imágenes tomadas por la cámara de un *smartphone* BQ Aquaris X Pro (cámara de 12 MP), aunque también hay otras tomadas con otros *smartphones*, como por un Samsung Galaxy S7 o un iPhone X, entre otros. Este *dataset* contiene imágenes tomadas en: San Vicente del Raspeig, San Juan de Alicante, Elche, Alicante, Valencia, Jumilla, Murcia, Blanca y Albacete.

Las imágenes se capturaron desde diferentes ángulos, pero siempre de frente a un

paso de peatones con un semáforo peatonal (en verde y en rojo) como se observa en la Figura 6.1, repitiendo por lo general una foto para cada semáforo en cada uno de sus dos posibles estados. Por la noche, se hicieron distintas capturas con distintos valores de exposición a la luz como se aprecia en la Figura 6.2, pues a valores más bajos, la imagen que se tomaba de las siluetas de los semáforos eran más nítidas. Como se buscaba que el algoritmo aprendiese bajo múltiples condiciones, se utilizaron imágenes en la que la cámara estaba expuesta a fogonazos de luz, y otras en las que gracias a reducir el valor del parámetro de exposición, las siluetas de los semáforos se distinguían más nítidas a pesar de oscurecer los pasos de peatones.

(a) Paso con semáforo peatonal en verde. (b) Paso con semáforo peatonal en rojo.

Figura 6.1: Semáforos y pasos peatonales de día.

(a) Parámetro de exposición normal.

(b) Parámetro de exposición normal.

(c) Parámetro de exposición bajo.

(d) Parámetro de exposición bajo.

Figura 6.2: Semáforos y pasos peatonales de noche con nivel de exposición reducido.

Una vez capturadas un total de 1010 imágenes, de las cuales la mitad fueron tomadas durante el día y la otra durante la noche, se procedió a realizar un código escrito en lenguaje *Python* para reorganizarlas de manera aleatoria y asignarles un nombre ordenado; la prima imagen del *dataset* recibe un nombre de "00000.jpg", mientras que la última recibe el nombre de "01000.jpg". Este criterio para nombrar las imágenes es meramente por claridad.

Una ver mezcladas e identificadas se comenzó con la fase de etiquetado. Para esta fase se hizo uso de una aplicación de código abierto llamada OpenLabeling [1]. Antes de ponernos a etiquetar hay que indicarle a la aplicación cuáles son los nombres de las clases que nos interesa etiquetar; en este caso fueron "pasos_de_peatones","semaforo_verde" y "semaforo_rojo" (sin tildes).

Esta aplicación crea, para cada una de las fotos etiquetadas, un archivo de extensión *.txt* que contiene en cada una de sus líneas las etiquetas de cada uno de los objetos que hemos etiquetado previamente. Estas etiquetas se componen de un primer número entero que se corresponde con el identificador de la clase, seguido de las coordenadas en las que se encuentra el centro del *bounding box* que encierra al objeto, y finalizando con el ancho y alto de este *bounding box*:

<clase del objeto><centro X><centro Y><ancho><alto>

Hay que tener en cuenta que tanto los valores de las coordenadas como los del ancho y alto son valores comprendidos entre 0.0 y 1.0; el ancho y la coordenada 'X' se divide por el ancho total de la imagen, mientras que los valores del alto y de la coordenada 'Y' se dividen por el valor del alto de la imagen.

Una vez se dispuso de las 1010 imágenes bien etiquetadas, se llevó a cabo un reparto aleatorio de estas entre un *set* de entrenamiento, otro de validación y otro de test, quedando el reparto como:

- 600 imágenes para el *set* de entrenamiento.
- 200 imágenes para el *set* de validación.
- 200 imágenes para el *set* de test.

El total de objetos etiquetados que contiene este dataset es el que se muestra en el Cuadro 6.1.

[1] https://github.com/Cartucho/OpenLabeling

Objetos etiquetados	
Clase	**Etiquetas**
pasos_de_peatones	996
semaforo_verde	508
semaforo_rojo	513

Cuadro 6.1: Número de objetos que contiene el *dataset*

6.2. Reajuste del modelo personalizado

En este proyecto se hizo uso del repositorio original de YOLOv3 [2], cuyos códigos están personalizados para trabajar con una serie de *datasets* como COCO o Pascal VOC. En este apartado se hablará de una serie de modificaciones que fueron necesarias para que YOLOv3 trabajase correctamente con el *dataset* previamente elaborado y así obtener el modelo apropiado.

Este repositorio se guardó en una carpeta llamada 'darknet'; esto lo comento por el hecho de que a continuación se hablará de directorios específicos dentro de esta carpeta y considero útil referenciarla de alguna manera. A continuación, se dividirá en subapartados cada uno de los archivos que fueron necesarios crear [21].

6.2.1. Archivo de configuración del entrenamiento (yolov3-tfg.cfg)

El primer paso para crear nuestro modelo es configurar un nuevo *.cfg* [21] , que se trata de un archivo de configuración de la red donde se incluyen todos los parámetros importantes de entrenamiento. Para crear un nuevo *.cfg* es recomendable copiar el original (.../darknet/cfg/yolov3.cfg) e ir modificándolo.

Dentro de nuestro archivo *.cfg* que llamamos 'yolov3-tfg.cfg', los primeros parámetros que se modificaron fueron el tamaño del *batch* y las *subdivisions* en las que se dividiría este durante el proceso de entrenamiento, quedando de la siguiente manera:

```
[net]
# Testing
# batch=1
# subdivisions=1
# Training
```

[2]https://github.com/pjreddie/darknet

```
batch=64
subdivisions=8
```

El hecho de determinar un tamaño de *batch* se debe a que no es práctico utilizar todas las imágenes del conjunto de entrenamiento a la vez para actualizar los pesos; es mejor hacer uso de un pequeño lote, en este caso de 64, para actualizar los parámetros de la red. En cuanto a las subdivisiones, permite procesar de manera fraccionada el *batch* de imágenes cuando no se dispone de una GPU con suficiente memoria para hacerlo con el *batch* completo. Para obtener el mejor valor para el parámetro de *subdivisions* es recomendable ir probando con múltiplos de dos hasta que deje de producirse un error por memoria insuficiente en la GPU; es este caso el mejor valor fue de 8. También es necesario destacar que durante la fase de test, tanto *batch* como *subdivisions* se establecen como 1.

Los siguientes parámetros a tener en cuenta en este archivo son los de *width*, *height* y *channels*, que definen el tamaño al que se redimensionarán las imágenes previamente al entrenamiento y el numero de canales que serán procesados, que en este caso son los tres canales de RGB.

```
width=608
height=608
channels=3
```

En este caso se definió un tamaño para las imágenes de 608×608, ya que a mayor tamaño de las imágenes, menor es la pérdida de información y se obtienen mejores resultados en el entrenamiento, a pesar de que la duración del este se prolongue cuanto mayor sea la resolución.

Otros de los dos parámetros a definir son el *momentum* y el *decay*. El *momentum* se encarga de penalizar grandes cambios en los valores de los pesos entre iteraciones; como se indicó anteriormente, los pesos se actualizan a cada *batch* de imágenes, lo que provoca que las actualizaciones de los pesos fluctúen bastante, por lo este parámetro se hace necesario para prevenir una fluctuación descontrolada. Por otra parte, durante el entrenamiento existe la posibilidad de que la red sobreaprenda debido a que algunos pesos empiezan a tomar valores muy altos, ajustándose mucho a los datos de entrenamiento, y es aquí donde entra el *decay*, el cual es un término de penalización que previene que esto ocurra. Los siguientes valores son los recomendados, y se deberían modificar solo en caso de que durante el entrenamiento se obtuviera un sobreajuste indeseado.

```
momentum=0.9
decay=0.0005
```

Durante el entrenamiento, también es necesario controlar el parámetro de *learning rate*, que es la tasa de aprendizaje de parámetros que controla la agresividad con la que se debería aprender en base al *batch* de datos obtenidos; este valor suele estar comprendido entre 0.01 y 0.0001.

Al comenzar el entrenamiento, el algoritmo empieza sin ninguna información, por lo que el *learning rate* debe tomar un valor alto inicialmente. Conforme va avanzando el entrenamiento y cada vez se dispone de más y más información, es necesario que los pesos empiecen a variar de manera menos agresiva, es decir, el *learning rate* debe ir reduciéndose con el tiempo. Para que esto se cumpla se hace uso del parámetro *policy*, que es nuestra política de reducción de la tasa de aprendizaje a partir del parámetro *steps*, cuyo valor es el número de iteraciones que deben pasar hasta que se vaya a comenzar a reducir el *learning rate*. A partir de la interacción que definamos, el *learning rate* se irá multiplicando por el valor que fijemos en *scales*. Es posible asignar varios valores para *steps* y *scales*.

A pesar de que es cierto que el *learning rate* debe ser alto al principio, también se ha demostrado empíricamente que la velocidad del entrenamiento aumenta si tenemos una tasa más baja durante un corto periodo de tiempo al principio; esto es controlado por el parámetro *burn-in*.

En nuestro archivo de configuración asignamos los siguientes parámetros, que significan que nuestro *learning rate* al principio adoptará un valor bajo, controlado por un *burn-in*, pero que a partir de la iteración 4800 (indicado en *steps*) comenzará a ser multiplicado por 0.1, y posteriormente en la iteración 5400 volverá a repetirse este escalado.

```
learning_rate=0.001
policy=steps
steps=4800,5400
scales=.1,.1
burn_in=1000
```

Dentro de este archivo también hay una serie de parámetros que permiten modificar las imágenes originales en otras con diferentes rotaciones y transformaciones de color durante el proceso de entrenamiento para así potenciar el valor de nuestro *dataset*. Los parámetros modificable para esto son *angle*, *saturation*, *exposure* y *hue*. En nuestro archivo dejamos los parámetros por defecto.

```
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

El siguiente parámetro a modificar es el *max batches*, que es el número máximo de iteraciones que se deben ejecutar durante el proceso de entrenamiento. Para definir el valor óptimo para este parámetro se aconseja multiplicar por 2000 el número de clases que se desean detectar. Como en este caso queremos detectar tres tipos diferentes de objetos, el valor definido para *max batches* es de 6000.

Es necesario también indicar el número de clases con las que se va a trabajar (en nuestro caso tres) en las tres líneas en las que aparece el parámetro *classes* (líneas 610,696 y 783).

El parámetro final que hubo que calcular fue el número óptimo de filtros que se aplicarían en algunas de las capas de convolución. Este parámetro llamado *filters* hay que modificarlo tres veces en el texto, en las líneas 603, 689 y 776. El valor óptimo para este parámetro se obtiene como $filters = (classes + 5) \times 3$, que en nuestro caso es 24.

6.2.2. Archivo de datos (tfg.data)

En el archivo tfg.data se incluye la información sobre las especificaciones de nuestros objetos y algunas rutas importantes.

```
classes = 3
train  = /home/marcos/darknet/data/train_tfg.txt
valid  = /home/marcos/darknet/data/test_tfg.txt
names = /home/marcos/darknet/data/tfg.names
backup = backup
```

El parámetro *classes* necesita el número total de clases con las que nos manejamos. Los parámetros *train* y *valid* necesitan recibir la ruta en la que se encuentran unos archivos que contengas las rutas de cada una de las imágenes que componen el *set* de entrenamiento y validación, respectivamente. Es recomendable usar rutas absolutas para indicar la ubicación de estos archivos, al igual que para los siguientes parámetros.

La ruta que necesita *names* es la de un archivo que contenga los nombres de cada una de las clases en distintas líneas.

```
paso_de_peatones
semaforo_verde
semaforo_rojo
```

Por último, *backup* necesita que se le indique la dirección en la que debe guardar cada uno de los pesos que se irán generando durante el entrenamiento. Respecto a

cada cuantas iteraciones se guardan los pesos, por defecto, en el algoritmo se indica que cada cien iteraciones hasta llegar a las mil, y a partir de ahí los guarda cada mil iteraciones. Este criterio fue modificado para que se guardasen los pesos de la red cada época. Para ello se modificó la línea 138 del código detector.c alojado en la carpeta /darknet/examples.

6.3. Servidor Flask

Como se comentó en el apartado de objetivos, uno ellos es hacer uso del detector desde un dispositivo móvil mediante una aplicación Android. Para que esto fuese posible, la solución propuesta fue desarrollar un servidor web haciendo uso del *framework* de desarrollo Flask en lenguaje Python, el cual recibiera las imágenes, las procesara, y finalmente enviase una respuesta de si cruzar o no la calle.

Este servidor permanecerá constantemente a la espera de recibir una petición de tipo POST a la página *classify*. El método POST es utilizado para enviar una entidad de recurso específico. En este caso se aplica la petición POST para enviar una imagen.

El código del servidor contiene también el modelo de la red con los pesos óptimos para llevar a cabo la detección de los objetos en las imágenes recibidas. Una vez reconocidos los semáforos y pasos de peatones, el servidor posee la lógica necesaria para determinar si se puede o no cruzar la calle, así como para descartar las detecciones de probabilidad inferior al 0.8. Su conclusión final la enviará al cliente que envió la petición POST en un fichero de texto en formato JSON.

La lógica de la que hablamos consiste en una función que recibe los nombres de los objetos que detecta el modelo y su probabilidad de acierto. En función de los objetos que detecte y si su probabilidad es superior a 0.8, pone una serie de variables a *true*, es decir, si por ejemplo detecta un semáforo verde con probabilidad alta, asignará a su correspondiente variable un valor de *true*. Esta función devuelve una variable booleana con *true* en caso de que se haya detectado un paso de peatones y un semáforo peatonal en verde con probabilidades altas, y en todos los demás casos, devolverá *false*, contemplando la posibilidad de que el modelo no reconozca bien o que la fotografía no contenga alguno de los elementos necesarios.

Para que este servidor funcione en exteriores, es necesario hacer uso de una URL de acceso público. En este caso se utilizó una facilitada por la Universidad de Alicante. Para hacer pruebas en local, es posible hacer uso de la dirección IP de cualquier red WiFi y un puerto libre.

En la Figura 6.3 se representa el flujo de datos desde el *smartphone* al servidor web,

y del servidor al *smartphone*.

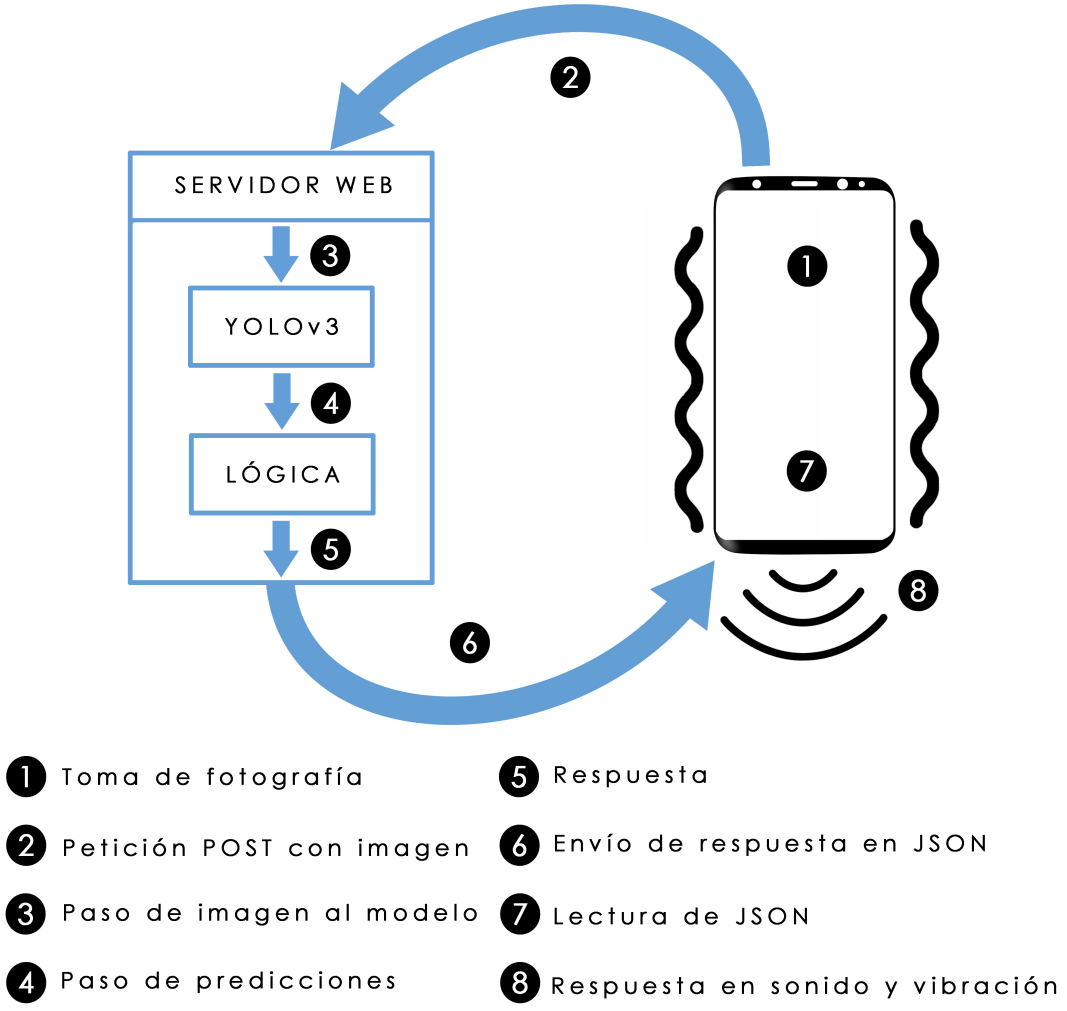


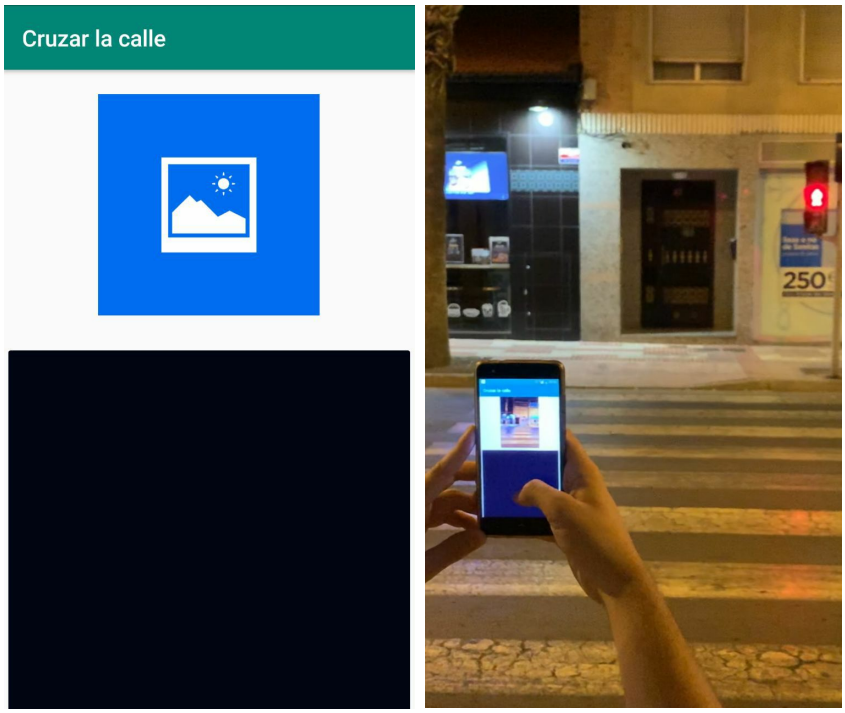
Figura 6.3: Flujo de datos durante el uso de la aplicación.

6.4. Desarrollo de aplicación Android

Para finalizar el proyecto, se desarrollará una aplicación con Android Studio aprovechando el clasificador resultante. Esta aplicación consiste en una interfaz de usuario sencilla y básica para smartphones, orientada a que una persona con una discapacidad visual sea capaz de utilizar. La interfaz está compuesta por dos partes principales; un botón y un *viewer*.

El botón se encuentra ubicado en la parte inferior de la pantalla del dispositivo,

ocupando la mitad de la pantalla y a una altura cómoda para ser pulsado por el dedo pulgar de la mano que sujete el smartphone, abriendo una aplicación de cámara que le permitirá al usuario capturar una imagen que será enviada y procesada posteriormente en un servidor, el cual contendrá el clasificador. El otro componente que conforma la interfaz, el *viewer*, es un elemento que muestra la imagen tomada por el usuario, con el propósito de que un familiar o conocido del usuario le pueda indicar si las imágenes se están tomando correctamente. Estos componentes se pueden observar en la captura de pantalla que se observa en la Figura 6.4a y en un caso de uso en la Figura 6.4b.



(a) Captura de pantalla de la app.

(b) Caso de uso de la app en la calle.

Figura 6.4: Imágenes de la app.

El funcionamiento de uso de la aplicación es el siguiente:

1. Una vez abierta la aplicación, pulsar en el botón ubicado en la parte inferior de la pantalla, abriendo así la aplicación de cámara.

2. Tomar foto de la calle que se pretende cruzar. Una vez tomada la imagen, esta se envía al servidor que la procesará.

3. Cuando el servidor procesa la imagen y devuelve una respuesta, la aplicación responderá de dos formas posibles:

- En caso de que la calle se pueda cruzar, es decir, en caso de que se detecte un paso de peatones y un semáforo peatonal en verde, la aplicación devolverá un sonido de dos tiempos junto a una doble vibración.

- En caso de que no se pueda cruzar la calle porque o bien no detecta semáforo o paso de peatones, o bien detecta el semáforo en rojo, la aplicación devuelve un sonido de un tiempo junto a una única vibración.

El propósito de haber añadido las vibraciones en las respuestas es debido a la posibilidad de que la persona también tenga alguna discapacidad auditiva, o que simplemente haya mucho tráfico y sea difícil escuchar el sonido que reproduce la aplicación.

7 Experimentación y resultados

En este capítulo se expondrán las diferentes pruebas que se llevaron a cabo con dos *datasets* diferentes, siendo estos Pascal Voc y el *dataset* propio. Se tratarán todos los problemas que surgieron durante la fase de preparación y ejecución del algoritmo, así como la evolución de los distintos entrenamientos con representaciones gráficas de una serie de valores que se mostraban por terminal, y finalizando con los resultados obtenidos a la hora de poner a prueba los algoritmos. También se describirán una serie de pruebas que se hicieron con el servidor web y se finalizará exponiendo los resultados de la aplicación final ante casos reales.

7.1. Experimentación con *dataset* Pascal Voc

El motivo de haber hecho uso de este *dataset* en lugar de pasar directamente al propio, se debe a que se buscaba una familiarización previa con YOLOv3. En esta fase se pudo aprender a como entrenar la red con un conjunto de datos distinto al que ya viene por defecto, así como a interpretar una serie de valores representativos que reflejan la evolución del aprendizaje en el tiempo tras cada iteración.

Todos los archivos necesarios para entrenar con Voc se obtienen directamente al instalar *Darknet*, a excepción del *dataset* Voc. A continuación se expondrán en distintos subapartados una serie de fases por las que hubo que pasar hasta obtener un entrenamiento exitoso.

7.1.1. Preparación del *dataset*

Para entrenar YOLOv3 con Pascal Voc, se siguieron los pasos descritos en el sitio web oficial de YOLO [15], donde se facilita un enlace a la descarga de Voc [1]. Para obtener todos los datos, se recomienda ejecutar los siguientes comandos dentro de un directorio en el que lo queramos alojar.

[1] https://pjreddie.com/projects/pascal-voc-dataset-mirror/

```
wget https://pjreddie.com/media/files/VOCtrainval_11-May-2012.
    tar
wget https://pjreddie.com/media/files/VOCtrainval_06-Nov-2007.
    tar
wget https://pjreddie.com/media/files/VOCtest_06-Nov-2007.tar
tar xf VOCtrainval_11-May-2012.tar
tar xf VOCtrainval_06-Nov-2007.tar
tar xf VOCtest_06-Nov-2007.tar
```

Con esto ya se obtienen todas las imágenes en un subdirectorio llamado *VOCdevkit*. El siguiente paso fue generar todas las etiquetas con el formato que necesita Darknet, tal y como se comenta en el Apartado 6.1. Para generar estas etiquetas, se facilita un código Python que lo hará por nosotros una vez lo ejecutemos.

```
wget https://pjreddie.com/media/files/voc_label.py
python voc_label.py
```

Tras ejecutar el código, se crearán los archivos con las etiquetas en los subdirectorios *VOCdevkit/VOC2007/labels/* y *VOCdevkit/VOC2012/labels/*. En el directorio observaremos ahora cinco archivos *.txt* que contienen los nombres de las imágenes del *dataset*: 2007_test.txt, 2007_train.txt, 2007_val.txt, 2012_train.txt y 2012_val.txt. Los nombres de estos archivos referencian el año y conjunto del *dataset* al que pertenecen. Darknet necesita un archivo *.txt* con todas las imágenes con las que queremos realizar el entrenamiento, por lo que se juntaron todos los nombres de las imágenes que contenían estos archivos en uno solo, a excepción del 2007_test.txt para poder usarlo como conjunto de testeo del modelo final ya entrenado. Para ello se usó el comando siguiente.

```
cat 2007_train.txt 2007_val.txt 2012_*.txt > train.txt
```

Con esto ya se concluiría la parte de preparación del conjunto de datos, y restaría modificar el contenido del archivo llamado *cfg/voc.data*, en el que solo hay que modificar las rutas en las que se encuentran los archivos con los nombres de la imágenes del conjunto de entrenamiento y validación.

```
1 classes= 20
2 train  = <ruta-a-voc>/train.txt
3 valid  = <ruta-a-voc>/2007_test.txt
4 names = data/voc.names
5 backup = backup
```

Para finalizar y antes de dar comienzo al entrenamiento, se requiere de unos pesos para inicializar de primeras el modelo. Estos pesos iniciales se recomienda que sean los preentrenados por los desarrolladores con el *dataset* ImageNet [2].

A partir de este punto, se pasó a realizar el primer intento de entrenamiento de la red, y fue aquí cuando comenzaron una serie de problemas que hubo que solventar.

7.1.2. Primeros entrenamientos fallidos

El primer entrenamiento se hizo con el equipo que integra la Nvidia GeForce GTX 960M del que se habla en el apartado 5.1.1. Para ejecutar el entrenamiento se utilizó el siguiente comando:

```
./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg
    darknet53.conv.74
```

El algoritmo estuvo entrenando durante más de ocho horas, mientras que en la terminal, a cada iteración, se imprimía un mensaje como el que se muestra a continuación, en el que se puede observar como la mayoría de los parámetros que se muestran poseen un valor Not-A-Number (NaN). También se observa como en la última se indica que se están guardando los pesos calculados en un fichero llamado *backup*.

```
Loaded: 0.000037 seconds
Region 82 Avg IOU: nan, Class: nan, Obj: nan, No Obj: nan, .5R: 0.000000, .75R:
    0.000000,  count: 1
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
    count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
     count: 0
50198: -nan, -nan avg, 0.000010 rate, 0.757152 seconds, 50198 images
Loaded: 0.000032 seconds
Region 82 Avg IOU: nan, Class: nan, Obj: nan, No Obj: nan, .5R: 0.000000, .75R:
    0.000000,  count: 1
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
    count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
     count: 0
50199: -nan, -nan avg, 0.000010 rate, 0.755875 seconds, 50199 images
Loaded: 0.000038 seconds
Region 82 Avg IOU: nan, Class: nan, Obj: nan, No Obj: nan, .5R: 0.000000, .75R:
    0.000000,  count: 1
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
    count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: nan, .5R: -nan, .75R: -nan,
     count: 0
50200: -nan, -nan avg, 0.000010 rate, 0.757097 seconds, 50200 images
Saving weights to backup/yolov3-voc.backup
```

[2]https://pjreddie.com/media/files/darknet53.conv.74

Estas salidas se corresponden a las últimas del entrenamiento, por lo que el siguiente paso era comprobar si el entrenamiento había ido bien, aunque a priori con tantos valores NaN no parecía que fuese a haber ningún éxito. Para testear el modelo se le pasó una imagen que contenía un gato y los pesos que se había calculado durante el entrenamiento. El comando que se utilizó fue el siguiente:

```
./darknet detector test cfg/voc.data cfg/yolov3-voc.cfg
   backup_mal/yolov3-voc_final.weights voc_data/gato.jpeg
```

Los resultados fueron negativos, pues el algoritmo no fue capaz de detectar el gato ni los demás objetos de una serie de imágenes posteriores que se le pasaron, lo que sirvió para comprobar que definitivamente algo había ido mal.

Todos estos pasos descritos hasta ahora se replicaron en el equipo de Rovit que incorpora la GPU Nvidia Quadro P6000, obteniendo los mismos valores de NaN durante el entrenamiento. Para corregir esto se dio paso a revisar el archivo de configuración del modelo llamado 'yolov3-voc.cfg', donde se comprobó que el tamaño de *batch* y *subdivisions* estaba a 1 por defecto. Para comprobar si esto tenía algo que ver con la aparición de los NaN, se hizo una prueba con entrenamiento con un *batch* cuatro imágenes y dos *subdivision*, que dio como resultado los primeros valores lógicos durante el entrenamiento.

Una vez resuelto este problema, se vio necesario averiguar qué eran cada uno de los parámetros que se mostraba por terminal durante el entrenamiento [22] [23] para posteriormente construir un gráfico, y observar y entender la evolución de este. En el siguiente fragmento se observa la salida correspondiente a dos subdivisiones de un conjunto de imágenes de entrenamiento.

```
Region 82 Avg IOU: 0.730200, Class: 0.778006, Obj: 0.629192, No Obj: 0.008634, .5R:
    1.000000, .75R: 0.545455,  count: 11
Region 94 Avg IOU: 0.749213, Class: 0.710076, Obj: 0.573380, No Obj: 0.000941, .5R:
    1.000000, .75R: 0.375000,  count: 8
Region 106 Avg IOU: 0.750606, Class: 0.393133, Obj: 0.004155, No Obj: 0.000015, .5R:
    1.000000, .75R: 1.000000,  count: 1
Region 82 Avg IOU: 0.801039, Class: 0.889127, Obj: 0.562734, No Obj: 0.006938, .5R:
    1.000000, .75R: 0.800000,  count: 10
Region 94 Avg IOU: 0.834949, Class: 0.617838, Obj: 0.988017, No Obj: 0.000556, .5R:
    1.000000, .75R: 1.000000,  count: 2
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -
    nan,  count: 0
4800: 0.805576, 0.955536 avg, 0.001000 rate, 5.062492 seconds, 307200 images
```

Comenzaremos definiendo qué indica la primera línea de esta salida, describiendo el significado de los parámetros más relevantes:

- Region 82 Avg IOU: indica el IOU promedio del conjunto de imágenes de la actual subdivisión. Se observa que hay tres regiones (82, 94 y 106), las cuales

corresponden a las tres escalas a las que se realizan las predicciones. En algunos casos puede aparecer un valor NaN, pero esto no tiene por qué significar que se trate de un mal entrenamiento, a no ser que aparezca de manera permanente en todas las iteraciones y en todas las escalas. Esto se puede deber a la pérdida de información durante el proceso de reducción del mapa de características.

- Class: su valor es el promedio de las probabilidades de los verdaderos positivos del subconjunto de imágenes.
- Obj: refleja el promedio de confianza que hay en que los *bounding box* que predice el modelo contengan realmente un objeto. Se considera una detección cuando el valor de IOU para cada objeto es igual o superior al 50 %.
- No Obj: es la misma idea que con 'Obj', pero aquí se esperan valores bajos, que quieren decir que la red está segura de que los *bounding box* que genera caen, por ejemplo, en el cielo, en la carretera o en el agua.
- Avg Recall: es el valor de *recall* explicado en el capítulo de *Metodología*. Refleja cuantos de los objetos de la subdivisión que debían ser detectados lo han sido.
- Count: número total de objetos que fueron detectados en la subdivisión.

La última línea del fragmento ofrece información más general de cómo está hasta ahora el entrenamiento:

- 4800: indica el número de *batches* usados para entrenar hasta el momento.
- 0.805576: error acumulado. Cuanto menor sea este valor, mejores resultados. Se puede utilizar como una referencia para detener el entrenamiento. Si se observasen valores NaN en este campo, el entrenamiento estaría sufriendo algún fallo.
- 0.955536 avg: error promedio del *batch* actual. Cuanto menor sea este valor, mejor entrenamiento. También se puede tomar como referencia para detener el entrenamiento. Si se observasen valores NaN en este campo, el entrenamiento estaría sufriendo algún fallo.
- 0.001000 rate: *learning rate* utilizado para actualizar los pesos en el *batch* actual.
- 5.062492 seconds: tiempo que ha transcurrido para entrenar con el *batch*.
- 307200 images: número de imágenes con el que se ha entrenado de momento. Con este valor se pueden calcular las épocas transcurridas.

Tras una serie de entrenamientos y testeos fallidos con distintos tamaños de *batch*

y *subdivision*, se logró determinar unos buenos valores siguiendo una serie de consejos vistos en un sitio web en el que se tratan varios temas al respecto de YOLO [21]. Una vez se obtuvo más información respecto a qué podía estar pasando, se optó por modificar el criterio con el que se guardaban los pesos; mientras antes se guardaban cada cien iteraciones hasta 900, y a partir de ahí cada diez mil, ahora se guardarían cada época. Esto permitiría poder tener un mejor control del entrenamiento, pudiendo comprobar si en algún momento se produce un sobreentrenamiento, ya que se dispondría de un conjunto de datos cuya distancia en el tiempo ahora es menor.

7.1.3. Entrenamiento exitoso

Para este entrenamiento se fijó un *batch* de 64 imágenes y un *subdivision* de 8, pues era la combinación que mejor se ajustaba a la GPU Nvidia Quadro P6000. Antes de dar comienzo al entrenamiento, en el código ubicado en *'darknet/examples/detector.c'* se modificó en la línea 138 el criterio para que se guardasen los pesos cada época. Una época correspondería a procesar las 16.551 imágenes que contiene el *dataset* de VOC, y teniendo en cuenta que las imágenes se usarán de 64 en 64 cada iteración, una época correspondería a 258 iteraciones.

Tras estas modificaciones, se dio paso a la ejecución del entrenamiento, que permanecería durante 12 horas, completando un total de 9132 iteraciones (35 épocas completas). Para poder guardar los valores que se imprimían por terminal durante el entrenamiento en un archivo *.txt*, se hizo uso del siguiente comando:

```
./darknet detector train cfg/voc.data cfg/yolov3-voc.cfg
   darknet53.conv.74 \textbf{2>&1 | tee output.log}
```

Tras la obtención de este archivo, se comenzó a programar un código capaz de recoger la información de la salida del entrenamiento a cada época. Este era capaz de plotear la información del *total loss*, *avg loss*, IOU , *class*, *obj*, *no obj* y *count*. A excepción de los errores, los demás valores se plotean para las tres distintas escalas, además del valor medio de estas. En la Figura 7.1 se observa como para representar estos cuatro valores se hace uso de diferentes colores; el azul corresponde a valores de la escala 82, el verde para la 94 , el amarillo para la 106, y el negro para la media de las tres.

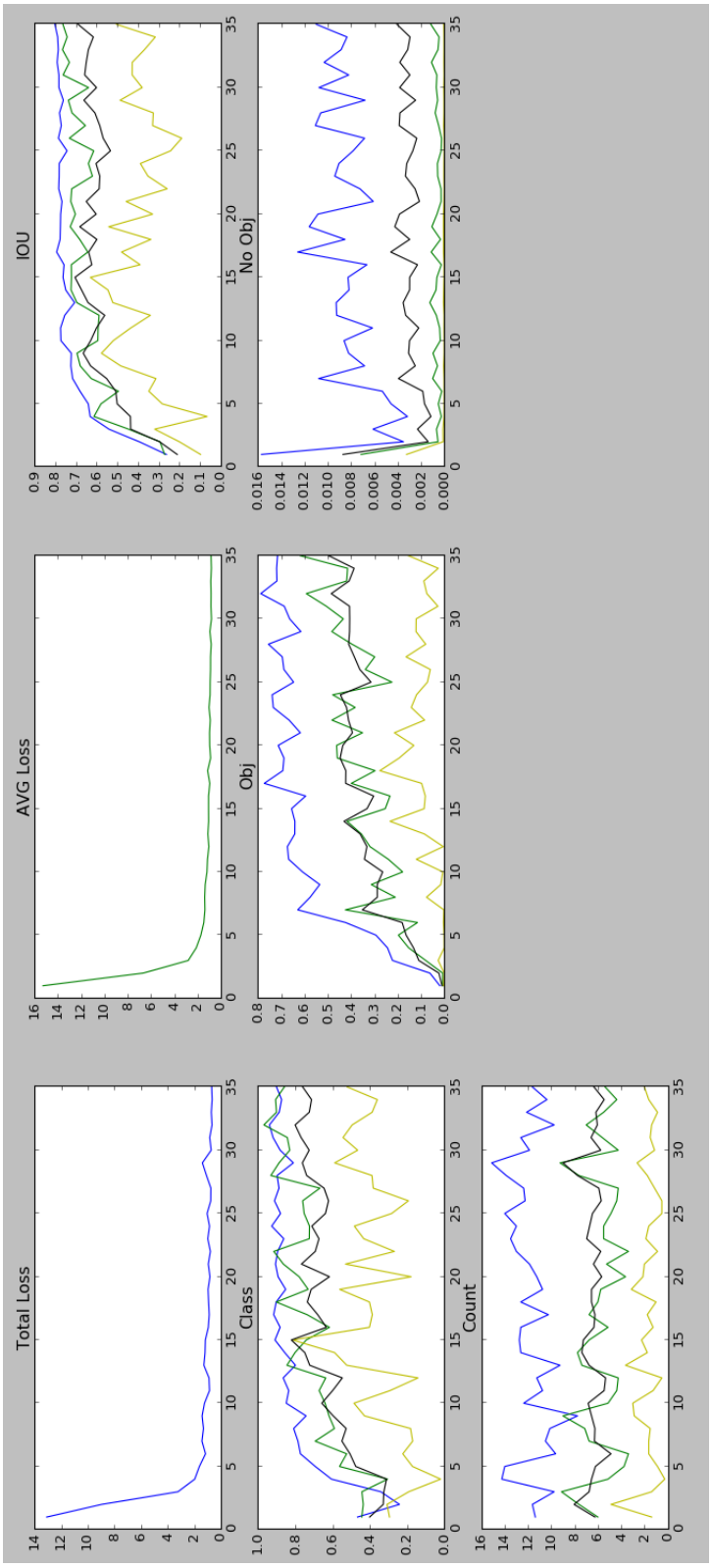


Figura 7.1: Gráficas de los distintos valores mostrados durante entrenamiento.

En esta Figura 7.1 se puede observar como a priori el entrenamiento se ha completado con éxito. Esto se deduce viendo por ejemplo las gráficas de los errores, donde se observa que sus valores decrecen rápidamente y se estabilizan por debajo de la unidad. Otro aspecto es que el valor de IOU medio en la última época toma un valor de 0.7, lo que es bastante prometedor. También se observan valores altos para las demás predicciones. Estos resultados apuntaban a que este podría ser un buen entrenamiento. Para comprobarlo, se hizo uso del comando de *recall*, pero surgieron una serie de problemas en este paso.

Al ejecutar el *recall* tras haber cambiado previamente los valores de *batch* y *subdivision* a '1', con los pesos obtenidos en la primera época se obtenían valores de *recall* e IOU razonables. El problema aparecía al ejecutar el código con los pesos de las demás épocas. Lo que se observaba era una especie de cuenta atrás en ambos valores, es decir, los valores se iban decrementando consecutivamente a cada nueva imagen a la que se le aplicaba el algoritmo. Para dar solución a esto se siguieron las recomendaciones descritas en [24], que consistían básicamente en modificar la linea 546 del código *detector.c*.

```
546     // for(k = 0; k < l.w*l.h*l.n; ++k){
547     for(k = 0; k < nboxes; ++k){
548         float iou = box_iou(dets[k].bbox, t);
549         if(dets[k].objectness > thresh && iou > best_iou){
550             best_iou = iou;
...
```

Con este problema solucionado y ante la situación de que ya se podía obtener los distintos valores de *recall* e IOU con los pesos obtenidos en las distintas épocas, se empezó a desarrollar un código para grafiarlos y poder observar la evolución de la mejora en el reajuste de los pesos y si se había producido algún tipo de sobreentrenamiento. Para poder procesar esta información, se necesitaba guardar previamente la salida que se mostraba por terminal para cada uno de los pesos. Para facilitar el proceso de ejecución del *recall* se elaboró un archivo *.sh* ubicado directamente en el directorio de *darknet*, el cual contenía los comandos necesarios para cada conjunto de pesos. De esta manera fue menos tendioso que ir uno por uno ejecutando cada *recall*. A continuación se muestran los comandos para los dos primeros conjuntos contenidos en el *.sh*.

```
./darknet detector recall cfg/voc.data cfg/yolov3-voc.cfg
   backup/yolov3-voc_258.weights 2>&1 | tee logs/258_recall.
   log

./darknet detector recall cfg/voc.data cfg/yolov3-voc.cfg
   backup/yolov3-voc_516.weights 2>&1 | tee logs/516_recall.
   log
```

```
...
```

Una vez desarrollado el código para extraer los valores necesarios de los archivos *.log*, se obtuvo el resultado mostrado en la Figura 7.2, en el que se aprecian valores altos tanto para *recall* como para IOU en las últimas épocas. También se puede apreciar como no se ha producido ningún tipo de sobreentrenamiento, pues no se observa ningún valle pronunciado en ninguna de las gráficas que denote que el algoritmo hubiera comenzado a sobreajustar los pesos en base a los datos de entrenamiento.

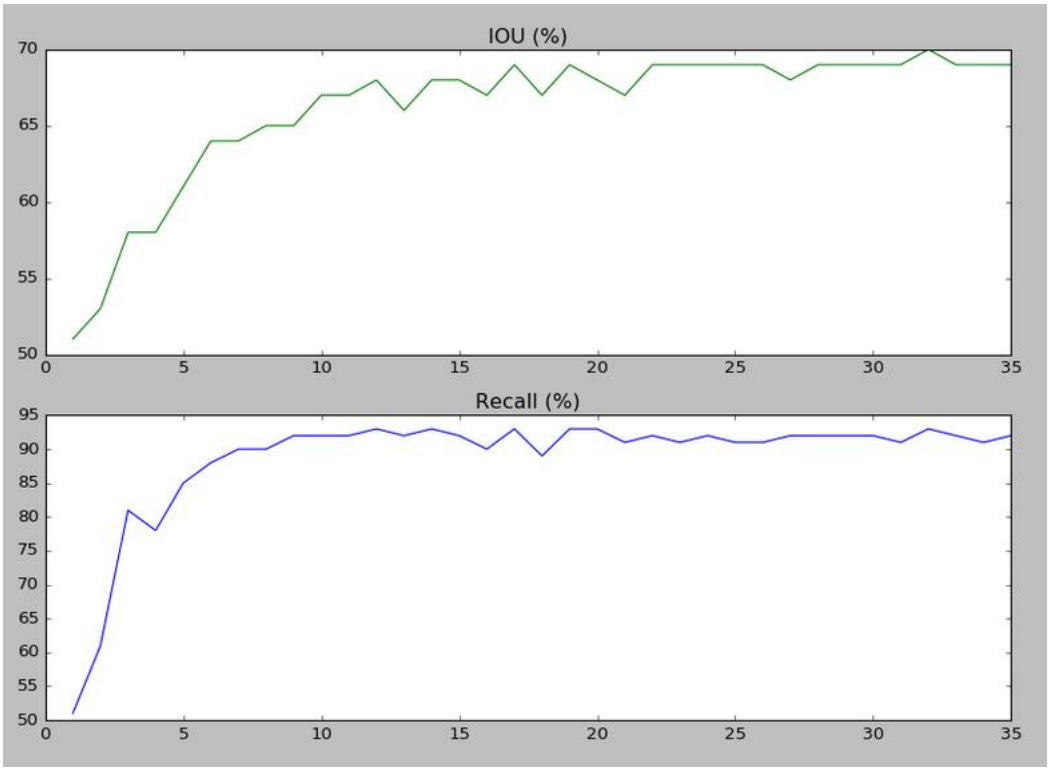


Figura 7.2: Gráficas de los distintos valores de *recall* e IOU obtenidos con los pesos calculados en cada época.

7.2. Experimentación con *dataset* propio

La modificación personalizada del modelo ya se explica en el Apartado 6.2 , por lo que ahora se va a tratar tan solo las observaciones tomadas durante y después del entrenamiento de dicho modelo. En esta fase de experimentación también se hizo uso de los códigos descritos anteriormente para grafiar los valores de salida del entrenamiento

y los de *recall*.

Una vez se tuvo el modelo bien estructurado en el equipo de Rovit con la Quadro P6000, se dio lugar a la ejecución del entrenamiento con el comando siguiente , donde se le indica que guarde todas las salidas en un fichero *.log* e inicialice con los pesos originales de Darknet53. Este entrenamiento se mantuvo durante un total de 455 épocas.

```
./darknet detector train data/tfg.data cfg/yolov3-tfg.cfg
   darknet53.conv.74 2>&1 | tee entrenamiento_1.log
```

Tras finalizar el entrenamiento, se pasó a grafiar los valores de salida para obtener una representación gráfica de cómo había ido el entrenamiento. Estas gráficas se observan en la Figura 7.3, donde el color azul corresponde a valores de la escala 82, el amarillo a la 94, el verde a la 106 y el negro a la media entre las tres. Se puede comprobar que los errores tienden a reducirse verticalmente al poco de comenzar el entrenamiento. También se aprecia cómo los valores de las predicciones son muy altos, aunque para la escala de 106 se reducen bastante. Esto puede ser debido a una pérdida de información de los semáforos en los mapas de características, ya que debido a la distancia, estos eran bastante pequeños en las imágenes. En cuanto a los valores de IOU, también se obtuvieron valores elevados, incluso llegando la media de las tres escalas al 0.7.

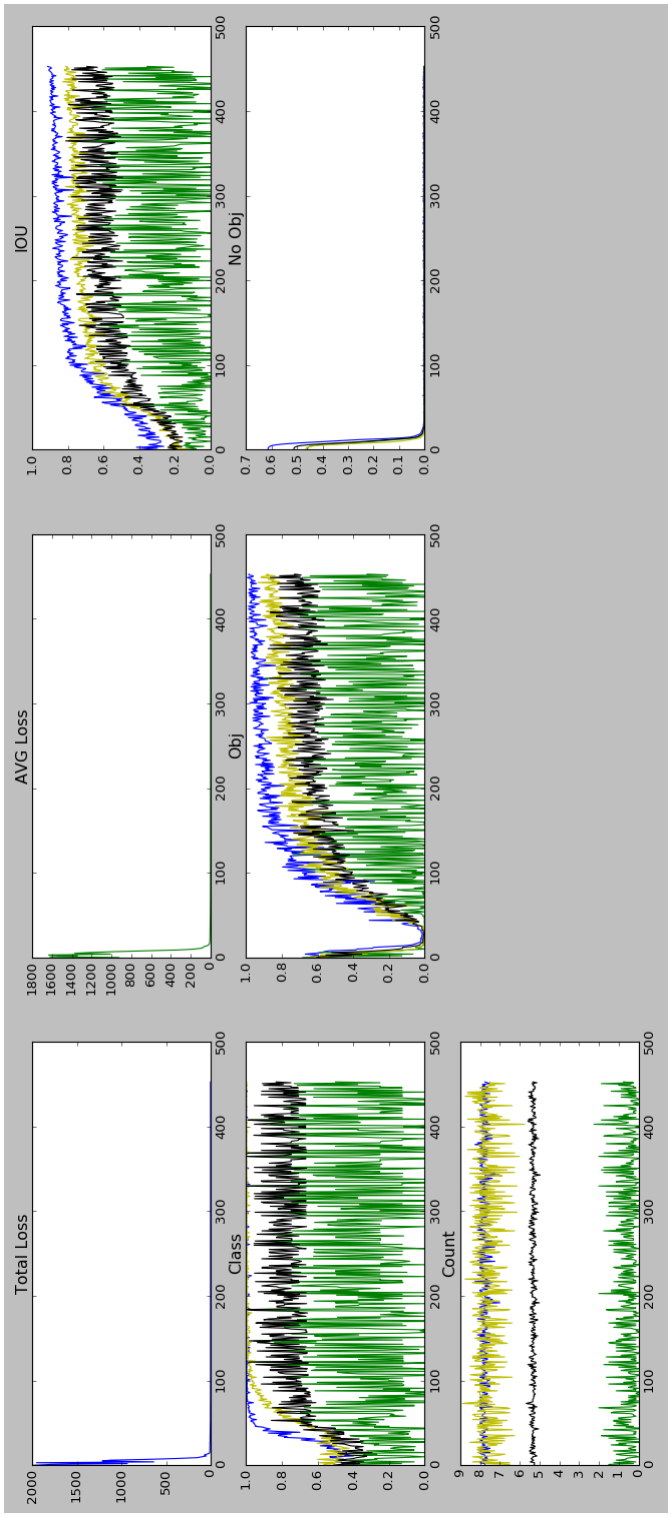


Figura 7.3: Gráficas de los distintos valores de salida durante el entrenamiento del modelo personalizado.

El siguiente paso fue hacer uso del comando de recall, pero surgió un error que hubo que corregir. Durante la ejecución del comando se observaba que no se cogían las 200 imágenes que correspondían al conjunto de test, por lo que hubo que investigar en el código *detector.c* de donde se estaban cogiendo las imágenes. La solución fue modificar la línea 497, perteneciente a la función *'validate_detector_recall'*, donde se indicó lo siguiente:

```
list *plist = get_paths("data/test_tfg.txt");
```

Con este contratiempo solucionado, se pasó a generar un archivo *.sh* con los comandos necesarios para ejecutar el *recall* de todos los pesos calculados de las diferentes épocas y guardar sus salidas. Tras obtener todos los *.log*, se le pasaron al código diseñado para grafiar los valores de *recall* e IOU, y los resultados fueron los mostrados en la Figura 7.4. En estas gráficas se observa cómo no se produce ninguna desmejora pronunciada en estos valores, lo que quiere decir que el sistema no se ha sobreajustado en ningún momento a los datos del conjunto de entrenamiento. Los valores medios finales fueron de 92 % para *recall* y de 75 % para IOU.

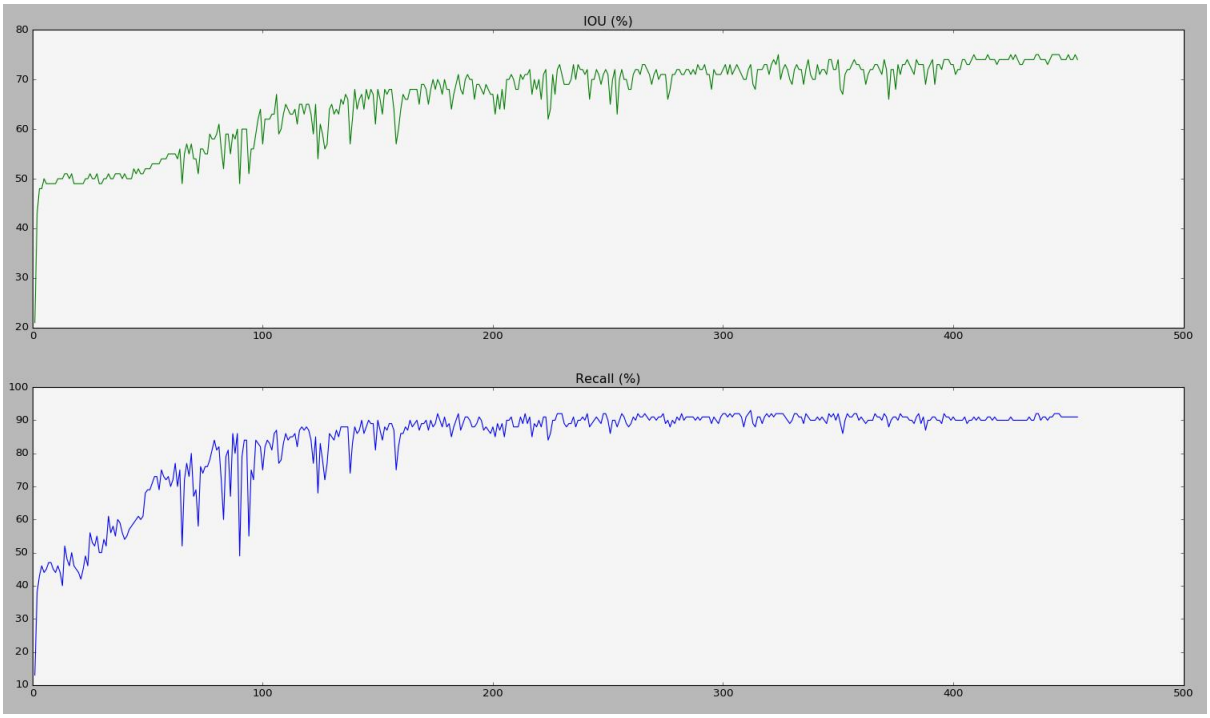


Figura 7.4: Gráficas de los valores de *recall* e IOU tras aplicarles los pesos de las diferentes épocas al conjunto de imágenes de test.

Con estos resultados, se decidió que el siguiente paso sería realizar un código al que pasarle una serie de vídeos, que los descompusiera en *frames* y analizase cada uno de

estos buscando los pasos de peatones y los semáforos. Para cada uno de los objetos, el código dibujaría un *bounding box* indicando la clase a la que corresponden, y finalmente juntaría todos estos *frames* con los *bounding boxes* en un vídeo.

Para esto se aprovechó el código *detector.py* que ya venía incluido con la descarga de Darknet. Este código contenía un ejemplo de detección muy sencillo y específico para los pesos por defecto de Darknet53. A la hora de adaptarlo hubo que modificar algunas rutas, y para evitar una serie de errores, se indicaron como absolutas. Estas rutas son necesarias para cargar el modelo de la red (*.cfg*), indicarle los pesos y el conjunto de datos (archivo *.data*) que se van a utilizar.

Tras aplicar este código a varios vídeos, se obtuvieron buenos resultados, en los que no se perdía la detección de los objetos deseados y no se cometían errores a la hora de diferenciar los semáforos entre verde y rojo, además que no se clasificaban los semáforos de tráfico como semáforos de peatones. En la Figura 7.5 se observan dos casos de *frames* a los que se les aplicó este código.

Figura 7.5: Semáforos y pasos peatonales detectados con *detector.py*

Ante los buenos resultados que mostró este modelo, se decidió compararlo con las detecciones que hacía el modelo original con el *dataset* COCO, pues también detectaba semáforos, pero como se pudo observar en los vídeos tras el proceso de detección, no hacía distinción de si el semáforo estaba encendido o apagado, ni diferenciaba el estado rojo del verde, incluso no diferenciaba entre semáforo de peatones y de tráfico. Esto era lo que se esperaba, pero la diferencia más relevante respecto a nuestro modelo personalizado, es que las detecciones en los casos de noche eran bastante malas. Para hacer una comparativa más visual, se optó por juntar en un único vídeo los resultados de las detecciones de los dos modelos, el cual se puede ver en el canal de YouTube de Rovit [3]. A continuación, en la Figura 7.6 se observa un caso de detección, por parte del modelo de COCO, que detecta un semáforo apagado, mientras el nuestro solo detecta el que nos interesa, es decir, el semáforo peatonal en rojo. También se puede comprobar en

[3]https://www.youtube.com/watch?v=R7CVWgt1SQQ

la Figura 7.7 como por la noche el modelo original pierde la detección de los semáforos, mientras el personalizado los detecta sin apenas dificultad. Tanto en la Figura 7.6 como en la Figura 7.7, la imagen izquierda pertenece al modelo de COCO y la derecha al propio.

Figura 7.6: Caso de detección de semáforos apagados con el modelo de COCO.

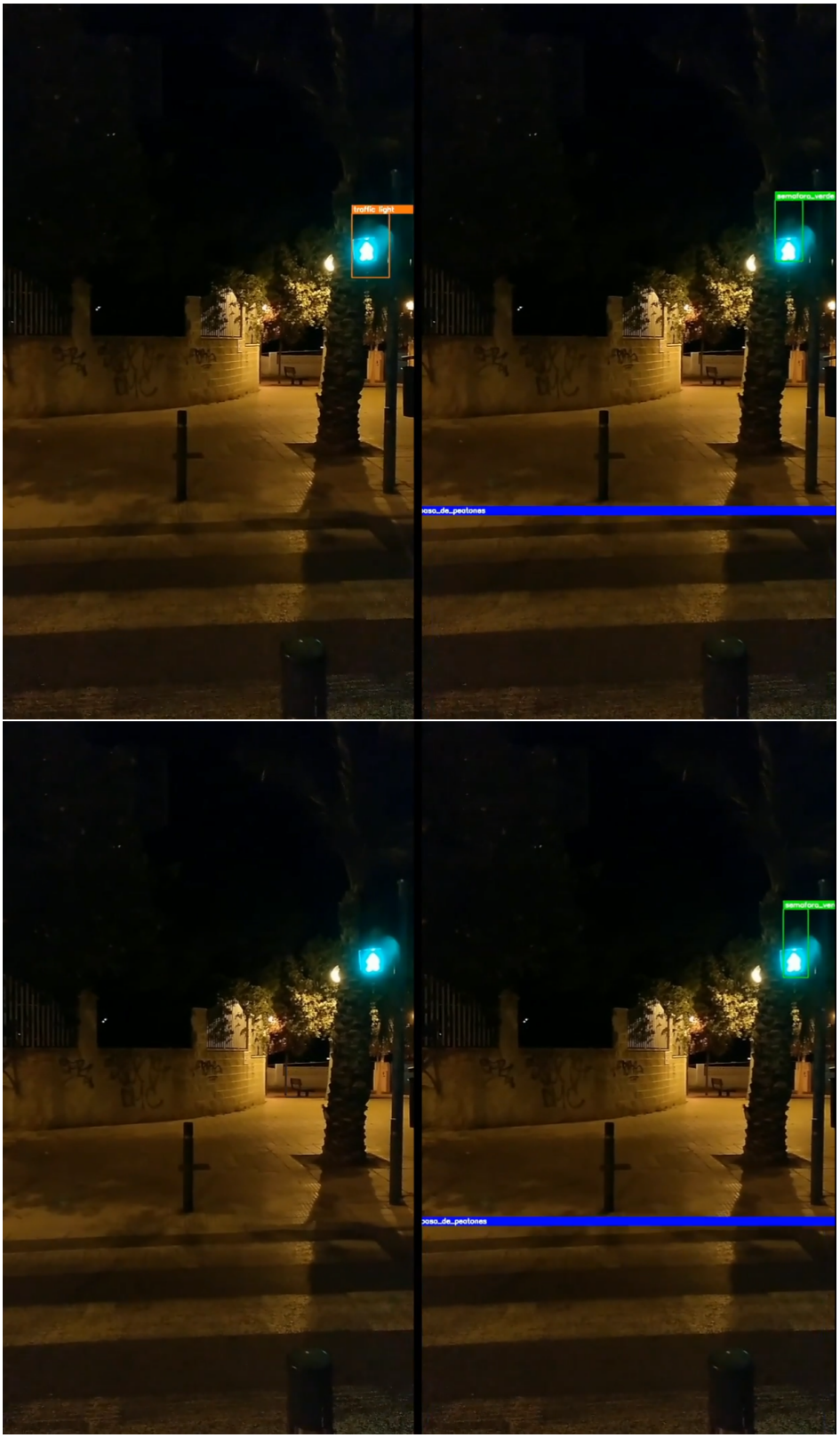


Figura 7.7: Caso de pérdida de la detección de un semáforo durante la noche con el modelo de COCO.

7.3. Experimentación con servidor Flask

La descripción de cómo se hizo este servidor web se encuentra en el Apartado 6.3, por lo que en este solo se tratarán algunas experiencias con el servidor durante su desarrollo y testeo.

De manera previa a utilizar la URL pública que se comenta en el Apartado 6.3, para hacer pruebas en local se utilizó una dirección IP '0.0.0.0', que en términos de servidor significa que todas las direcciones IPv4 están en el equipo local, y en caso de una ruta de entrada significa la ruta por defecto. Además de esta IP se utilizó el puerto 5000, ya que se trata de un puerto libre. Para generar el servidor fue necesario crear en primer lugar una aplicación Flask, y posteriormente ejecutarla indicando el *host* y el puerto como se muestra en el fragmento siguiente:

```
app = Flask(__name__) #se crea la app de flask
....
app.run(host='0.0.0.0', port=5000)
```

Tras ejecutar el servidor y cargar el modelo de la red, el servidor ya quedaría operativo para recibir peticiones y devolver respuestas en formato JSON. Para pasarle una imagen al servidor con intención de que la procesase y determinase si es un caso en el que se puede o no cruzar la calle, se utilizó un comando Curl que contenía la dirección de una imagen (recurso) y la dirección HTTP del *host*. Curl sirve para transferir datos desde o hacia un servidor. Un ejemplo de este comando es el siguiente:

```
curl -F "imagefile=@/home/marcos/00351.jpg" http
    ://0.0.0.0:5000/index
```

Este servidor permaneció así durante las primeras pruebas, pero una vez se comenzó el desarrollo de la app Android, hubo que utilizar otra dirección IP, en este caso la privada de un *router* de una red doméstica y el puerto 8080 para hacer las primeras pruebas, pues aun no se disponía de la URL que facilitó la Universidad de Alicante. Esta vez las peticiones POST con las imágenes se enviaban desde un *smartphone* conectado a la misma red que el servidor.

Una vez se terminó de desarrollar el sistema de comunicación de la app Android con el servidor Flask, fue cuando se solicitó la URL pública a la universidad para poder usar la aplicación en exteriores, pues así se podía usar la conexión a Internet mediante el 4G de un *smartphone* y probar la aplicación ante casos reales de la calle, como se dice en el Apartado 6.3.

7.4. Experimentación ante casos reales con la app Android

Para la última fase de experimentación, se descargó la aplicación en el mismo dispositivo BQ Aquaris X Pro con el que se había realizado el *dataset*, y tras esto se salió a la calle para hacer las primeras pruebas en entornos reales, tanto por la noche como por el día. Se realizó un total de 16 pruebas, cuyas respuestas fueron todas correctas. En el Cuadro 7.1 se indica en qué momento del día se llevó a cabo cada prueba, cuál fue el estado de los semáforos y el tiempo que hubo que esperar desde que se tomó la fotografía hasta que se recibió la respuesta en forma de sonido y vibraciones.

Pruebas en casos reales			
Nº Prueba	**Momento del día**	**Estado del semáforo**	**Tiempo de espera (s)**
1	noche	rojo	5.53
2	noche	verde	5.01
3	noche	rojo	5.29
4	noche	rojo	5.15
5	noche	verde	5.32
6	noche	rojo	5.38
7	noche	rojo	5.22
8	noche	verde	5.32
9	noche	verde	6.32
10	noche	rojo	6.15
11	día	verde	5.21
12	día	verde	6.48
13	día	rojo	5.28
14	día	rojo	4.18
15	día	verde	4.03
16	día	rojo	5.13

Cuadro 7.1: Información acerca de las pruebas en casos reales.

El tiempo medio de espera fue de 5.31 segundos, lo que es un tiempo razonable, pues hay que tener en cuenta que las imágenes que se tomaron y enviaron al servidor tenían un tamaño de 3024×4032 y haciendo uso de conexión 4G es normal tanta demora. Para comparar el tiempo de espera en la calle, se hicieron de nuevo unas pruebas con conexión WiFi, y en este caso el tiempo de espera medio fue de 3.23 segundos. Lo importante de estos resultados es el hecho de que la aplicación dio la respuesta correcta en todas las pruebas, pues el tiempo de espera se podría corregir modificando la resolución a la que tomar las imágenes y mejorando las comunicaciones.

A continuación, en las Figura 7.8 se observan los resultados de cuatro de estas pruebas. Son de destacar principalmente los resultados de dos de ellas; la prueba ocho mostrada en la Figura 7.8b y la prueba catorce mostrada en la Figura 7.8c. En la Figura 7.8b se observa cómo unos pocos metros más atrás del semáforo peatonal hay un semáforo de tráfico que el algoritmo no clasifica, lo que es un punto positivo ya que solo clasifica los semáforos que nos interesan. En la Figura 7.8c se presenta una situación en la que hay otros peatones ocultando gran parte del paso de peatones, pero el modelo es capaz de detectarlo utilizando tan solo una pequeña franja.

(a) Prueba 6.

(b) Prueba 8.

(c) Prueba 14.

(d) Prueba 15.

Figura 7.8: Caso de pérdida de la detección de un semáforo durante la noche con el modelo de COCO.

8 Conclusiones

Durante la elaboración de este proyecto nos hemos centrado en conseguir la manera de que darle autonomía a una persona con discapacidad visual a la hora de cruzar un paso de peatones señalizado con semáforos, y se podría decir que se ha conseguido una primera aproximación satisfactoria. La aplicación funciona correctamente y así lo han reflejado las pruebas que se han hecho con ella.

Se ha demostrado que el *dataset* que se elaboró es suficiente para obtener un buen entrenamiento con YOLOv3, y que para un número de clases pequeño, las 600 imágenes son más que suficientes para conseguir resultados satisfactorios, aunque no habría problema en ampliar el *dataset* con más casos de otras ciudades y condiciones de luz.

Es cierto que durante las pruebas hechas en la calle, el tiempo de espera hasta recibir la respuesta de si se podía cruzar era de 5.31 segundos de media. Esto no sería un problema grave ya que se podría reducir varios segundos configurando los parámetros de la cámara para capturar imágenes de menor tamaño y mejorando las comunicaciones.

También se puede afirmar que la interfaz final se adapta bien a las necesidades de una persona con discapacidad visual, pues se hace muy sencillo su uso sin tener que configurar nada. Para abrir la app, se puede hacer uso de la app Voice Access o incluso del Asistente de Google mediante voz. Una vez abierta, el usuario solo tiene que tocar la pantalla a la altura de su dedo pulgar con la mano que la agarra y tomar una foto de la calle, y tras ello solo queda esperar a recibir la respuesta por vibración y sonido de si puede cruzar.

Como líneas futuras se podría ampliar el dataset con casos de otras ciudades, pues por ejemplo, los semáforos de la ciudad de Madrid son diferentes a los que acostumbramos a ver en la Provincia de Alicante o en la Región de Murcia. También se podría investigar en cómo configurar los parámetros de la cámara desde la app Android para que por defecto, al utilizarla, las imágenes que se tomen y se envíen al servidor sean menos pesadas.

También se podría probar a entrenar alguna otra CNN con este *dataset* que sea capaz de correr en un *smartphone* o desarrollarla, e incluso probar los resultados que daría

la versión *tiny* de YOLOv3. Se podría también investigar la manera de hacer que las comunicaciones con el servidor fueran más rápidas. Finalmente se podría hacer más compleja la aplicación enviando vídeo al servidor en lugar de imágenes y que reconociera el estado del semáforo durante todo el tiempo que tarda el usuario en cruzar.

Bibliografía

[1] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep Residual Learning for Image Recognition. *arXiv*, 2015, 10 de diciembre.

[2] Joseph Redmon y Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv*, 2018.

[3] Enrique Pérez. Accesibilidad en Android: 21 aplicaciones para personas ciegas o con problemas de visión. https://www.xatakandroid.com/aplicaciones-android/accesibilidad-en-android-21-aplicaciones-para-personas-invidentes, 2017, 25 octubre. Fecha de última consulta: 2019, 21 de marzo.

[4] Deep Learning. http://deeplearning.net/. Fecha de última consulta: 2019, 25 de junio.

[5] Jason Brownlee. *Deep Learning with Python. Develop Deep Learning Models on Theano and Tensorflow Using Keras.* Machine Learning Mastery, 2017.

[6] Back-propagation. http://perso.wanadoo.es/alimanya/backprop.htm. Fecha de última consulta: 2019, 25 de junio.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[8] Raul E. Lopez Briega. Redes neuronales convolucionales con TensorFlow. https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/, 2016, 2 de agosto. Fecha de última consulta: 2019, 26 de junio.

[9] Siddharth Das. CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5, 2017, 16 de noviembre. Fecha de última consulta: 2019, 28 de junio.

[10] Yoshua Bengio y Patrick Haffiner Yann LeCun, León Bottoy. Gradiente-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 2018.

[11] Sik-Ho Tsang. Redes neuronales convolucionales con TensorFlow. https://medium.com/@sh.tsang/paper-brief-review-of-lenet-1-lenet-4-lenet-5-boosted-lenet-4-image-classification-1f5f809dbf17, 2018, 8 de agosto. Fecha de última consulta: 2019, 27 de junio.

[12] Karen Simonyan y Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv*, 2014, 6 de septiembre.

[13] Adrian Rosebrock. Imagenet: Vggnet, resnet, inception, and xception with keras, 2017, 20 de marzo.

[14] Sik-Ho Tsang. Review: ResNet — Winner of ILSVRC 2015 (Image Classification, Localization, Detection). https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8, 2018, 15 de septiembre. Fecha de última consulta: 2019, 29 de junio.

[15] Joseph Redmon. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/, 2013–2016. Fecha de última consulta: 2019, 3 de julio.

[16] Ross Girshick y Ali Farhadi Joseph Redmon, Santosh Divvala. You Only Look Once: Unified, Real-Time Object Detection. *arXiv*, 2016.

[17] Joseph Redmon y Ali Farhadi. YOLO9000: Better, Faster, Stronger. *arXiv*, 2017.

[18] Joseph Redmon. YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolo/. Fecha de última consulta: 3 de julio, 2019.

[19] La enciclopedia libre Wikipedia. Servidor web. https://es.wikipedia.org/wiki/Servidor_web, 2019, 21 de junio. Fecha de última consulta: 2019, 3 de julio.

[20] Google. Android Developer. https://developer.android.com/guide?hl=ES. Fecha de última consulta: 2019, 16 de junio.

[21] AlexeyAB. Yolo-v3 and Yolo-v2 for Windows and Linux. https://github.com/AlexeyAB/darknet. Fecha de última consulta: 2019, 10 de junio.

[22] Nils Tijtgat. Yolo-v3 and Yolo-v2 for Windows and Linux. https://timebutt.github.io/static/understanding-yolov2-training-output/, 2017, 7 de junio. Fecha de última consulta: 2019, 9 de marzo.

[23] Rafael Padilla. YOLO modifications. https://github.com/rafaelpadilla/darknet/blob/master/README.md#faq_yolo, 2018, 23 septiembre. Fecha de última consulta: 2019, 9 de marzo.

[24] z01nl1o02. Fix bug about recalling. https://github.com/z01nl1o02/darknet/commit/78ec26639dd5d6c77937325a99fb61801d31e858, 2018, 6 diciembre. Fecha de última consulta: 2019, 20 de marzo.